|  |
| --- |
| **Ingeniero en Computación**  **2018-2**  **Materia:** Algoritmos y Estructuras de Datos / Clave 12098  **Alumno:** Carlos Daniel Orozco Bravo  **Matrícula:** 351285  **Maestro:** Pedro Núñez Yépiz  **Práctica No.:** 1  **Tema – Unidad:** Archivos  **Ensenada Baja California a 8 de diciembre del 2018** |

|  |
| --- |
| 1. **INTRODUCCIÓN** |
| En esta practica desarrollaremos un programa de computadora en lenguaje c, capaz de manipular archivos binarios con estructura de datos. |

|  |
| --- |
| 1. **COMPETENCIA** |
| En esta práctica se busca que el alumno desarrolle la competencia, de poder hacer y manipular archivos binarios, desarrollado en un lenguaje c, combinando estructura de datos y archivos de texto. |

|  |
| --- |
| 1. **FUNDAMENTOS** |
| **Generalidades de archivos en c**  Muy a menudo necesitamos almacenar cierta cantidad de datos de forma más o menos permanente. La memoria del ordenador es volatil, y lo que es peor, escasa y cara. De modo que cuando tenemos que guardar nuestros datos durante cierto tiempo tenemos que recurrir a sistemas de almacenamiento más económicos, aunque sea a costa de que sean más lentos.  Durante la historia de los ordenadores se han usado varios métodos distintos para el almacenamiento de datos. Al principio se recurrió a cintas de papel perforadas, después a tarjetas perforadas. Más adelante se pasó al soporte magnético, empezando por grandes rollos de cintas magnéticas abiertas.  Hasta aquí, todos los sistemas de almacenamiento externo eran secuenciales, es decir, no permitían acceder al punto exacto donde se guardaba la información sin antes haber partido desde el principio y sin haber leído toda la información, hasta el punto donde se encontrase la que estábamos buscando.  Con las cintas magnéticas empezó lo que con el tiempo sería el acceso aleatorio a los datos. Se podía reservar parte de la cinta para guardar cierta información sobre la situación de los datos, y añadir ciertas marcas que hicieran más sencillo localizarla.  Pero no fué hasta la aparición de los discos magnéticos cuando ésta técnica llegó a su sentido más amplio. En los discos es más sencillo acceder a cualquier punto de la superficie en poco tiempo, ya que se accede al punto de lectura y escritura usando dos coordenadas físicas. Por una parte la cabeza de lectura/escritura se puede mover en el sentido del radio del disco, y por otra el disco gira permanentemente, con lo que cualquier punto del disco pasa por la cabeza en un tiempo relativamente corto. Esto no pasa con las cintas, donde sólo hay una coordenada física.  Con la invención y proliferación de los discos se desarrollaron los ficheros de acceso aleatorio, que permiten acceder a cualquier dato almacenado en un fichero en relativamente poco tiempo.  Actualmente, los discos duros tienen una enorme capacidad y son muy rápidos, aunque aún siguen siendo lentos, en comparación con las memorias RAM. El caso de los CD es algo intermedio. En realidad son secuenciales en cuanto al modo de guardar los datos, cada disco sólo tiene una pista de datos grabada en espiral. Sin embargo, este sistema, combinado con algo de memoria RAM, proporciona un acceso muy próximo al de los discos duros.  En cuanto al tipo de acceso, en C y C++ podemos clasificar los archivos según varias categorías:  Dependiendo de la dirección del flujo de datos:  De entrada: los datos se leen por el programa desde el archivo.  De salida: los datos se escriben por el programa hacia el archivo.  De entrada/salida: los datos pueden se escritos o leídos.  Dependiendo del tipo de valores permitidos a cada byte:  De texto: sólo están permitidos ciertos rangos de valores para cada byte. Algunos bytes tienen un significado especial, por ejemplo, el valor hexadecimal 0x1A marca el fin de fichero. Si abrimos un archivo en modo texto, no será posible leer más allá de un byte con ese valor, aunque el fichero sea más largo.  Binarios: están permitidos todos lo valores para cada byte. En estos archivos el final del fichero se detecta de otro modo, dependiendo del soporte y del sistema operativo. La mayoría de las veces se hace guardando la longitud del fichero. Cuando queramos almacenar valores enteros, o en coma flotante, o imágenes, etc, deberemos usar este tipo de archivos.  Según el tipo de acceso:  Archivos secuenciales: imitan el modo de acceso de los antiguos ficheros secuenciales almacenados en cintas magnéticas y  Archivos de acceso aleatorio: permiten acceder a cualquier punto de ellos para realizar lecturas y/o escrituras.  Según la longitud de registro:  Longitud variable: en realidad, en este tipo de archivos no tiene sentido hablar de longitud de registro, podemos considerar cada byte como un registro. También puede suceder que nuestra aplicación conozca el tipo y longitud de cada dato almacenado en el archivo, y lea o escriba los bytes necesarios en cada ocasión. Otro caso es cuando se usa una marca para el final de registro, por ejemplo, en ficheros de texto se usa el carácter de retorno de línea para eso. En estos casos cada registro es de longitud diferente.  Longitud constante: en estos archivos los datos se almacenan en forma de registro de tamaño constante. En C usaremos estructuras para definir los registros. C dispone de funciones de biblioteca adecuadas para manejar este tipo de ficheros.  Mixtos: en ocasiones pueden crearse archivos que combinen los dos tipos de registros, por ejemplo, dBASE usa registros de longitud constante, pero añade un registro especial de cabecera al principio para definir, entre otras cosas, el tamaño y el tipo de los registros.  Es posible crear archivos combinando cada una de estas categorías, por ejemplo: archivos secuenciales de texto de longitud de registro variable, que son los típicos archivos de texto. Archivos de acceso aleatorio binarios de longitud de registro constante, normalmente usados en bases de datos. Y también cualquier combinación menos corriente, como archivos secuenciales binarios de longitud de registro constante, etc.  En cuanto a cómo se definen estas propiedades, hay dos casos. Si son binarios o de texto o de entrada, salida o entrada/salida, se define al abrir el fichero, mediante la función fopen en C o mediante el método open de fstream en C++.  La función open usa dos parámetros. El primero es el nombre del fichero que contiene el archivo. El segundo es em modo que es una cadena que indica el modo en que se abrirá el archivo: lectura o escritura, y el tipo de datos que contiene: de texto o binarios.  En C, los ficheros admiten seis modos en cuanto a la dirección del flujo de datos:  r: sólo lectura. El fichero debe existir.  w: se abre para escritura, se crea un fichero nuevo o se sobrescribe si ya existe.  a: añadir, se abre para escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.  r+: lectura y escritura. El fichero debe existir.  w+: lectura y escritura, se crea un fichero nuevo o se sobrescribe si ya existe.  a+: añadir, lectura y escritura, el cursor se situa al final del fichero. Si el fichero no existe, se crea.  En cuanto a los valores permitidos para los bytes, se puede añadir otro carácter a la cadena de modo:  t: modo texto. Normalmente es el modo por defecto. Se suele omitir.  b: modo binario.  En ciertos sistemas operativos no existe esta distinción, y todos los ficheros son binarios.  En C++ es algo diferente, el constructor de las clases ifstream, ofstream y fstream admite los parámetros para abrir el fichero directamente, y también disponemos del método open, para poder crear el stream sin asociarlo con un fichero concreto y hacer esa asociación más tarde.  **buffers**  Ya hemos comentado que el acceso a los ficheros es lento, y lo es mucho, comparado con el acceso a memoria. Es por eso que, generalmente, no se accede a ficheros externos cada vez que se realiza una operación de lectura o escritura.  En su lugar, se mantiene una copia de una parte del fichero en la memoria, se realizan las operaciones de lectura/escritura que sea posible dentro de esa zona, y cuando sea necesario, porque alguna operación acceda a posiciones fuera de la zona almacenada, se vuelca esa zona al fichero y se lee otro tramo del fichero en memoria.  A estas zonas se le llaman buffers, y mejoran sensiblemente el acceso a los ficheros en lo que respecta a la velocidad.  Cuanto más grande es un buffer, mejor será el tiempo de acceso al fichero. En el caso ideal, el tamaño del buffer es mayor o igual que el del fichero, y todas las operaciones de lectura y escritura del fichero se realizan en memoria, de modo que sólo es necesario hacer una lectura del fichero y, si se ha modificado, una escritura.  Pero no todo son ventajas. Cuando se trabaja con buffers, las actualizaciones físicas del fichero están diferidas, en relación a las actualizaciones hechas por el programa, de modo que el fichero no siempre tiene una información actualizada.  Esto plantea dos problemas:  Si la aplicación termina de forma inesperada, por un error o por una avería, el contenido del buffer modificado no se almacenará en el fichero, y su estructura puede quedar corrupta y los datos inservibles.  Cuando un fichero deba ser accedido por varios usuarios de forma simultánea, se pueden presentar problemas de concurrencia. Por ejemplo, un usuario lee una parte del fichero en su buffer, y modifica su contenido. Mientras tanto, otro usuario, desde otra máquina, accede al mismo fichero, y a la misma zona, pero el contenido no está actualizado con las modificaciones realizadas por el primer usuario. El peligro es mayor si los dos están haciendo modificaciones en el mismo fichero, ya que es posible que las modificaciones realizadas por un usuario queden anuladas por las que ha hecho el otro.  El primer caso puede minimizarse, aunque no evitarse siempre, si se guarda el contenido del buffer antes de realizar operaciones potencialmente peligrosas. Aunque nada puede evitar la corrupción de ficheros en caso de avería.  El segundo caso requiere protecciones por parte del sistema operativo o de las aplicaciones que accedan a ficheros compartidos. Estas protecciones van desde las más simples, como la imposibilidad de que un segundo usuario acceda a un fichero abierto, hasta métodos más sutiles, como bloqueo de ficheros, o partes de ficheros. Estos bloqueos asignan una zona del fichero al primer usuario que lo solicite, e impiden a otros usuarios acceder a la misma zona, aunque no a otras.  **Ficheros que cambian de tamaño**  Por supuesto, todo lo explicado anteriormente es muy simple cuando sólo hacemos lecturas o cuando las escrituras no implican que el fichero deba ser más largos o más cortos.  Físicamente, los ficheros sólo pueden crecer por el extremo final. Esto es evidente en el caso de ficheros secuenciales, donde cualquier escritura que no se realize al final implica la sobrescritura de datos previos.  Las limitaciones físicas de los soportes de datos explican por qué no se pueden insertar datos en el interior de un fichero. En teoría se podría insertar un bloque de datos cuyo tamaño sea múltiplo del tamaño del cluster. De ese modo se podrían insertar nuevos clusters dentro del fichero. Pero esto es un caso muy especial, y nunca se hace.  Lo que se hace en realidad es mover el resto del fichero hacia adelante para dejar espacio para la nueva información, o en caso de borrar datos, mover hacia atrás, de modo que se sobrescriban los datos eliminados. De modo que si hay que añadir un byte en la primera posición de un fichero, esto implica que se ha de copiar todo el fichero. |

|  |
| --- |
| 1. **PROCEDIMIENTO** |
| hacer un programa que utilice un archivo binario. |

|  |
| --- |
| 1. **RESULTADOS Y CONCLUSIONES** |
| **Conclusiones:**  Se termino desarrollando un programa que leyera un archivo de texto al mismo tiempo de agregar los datos de una estructura de datos. |

|  |
| --- |
| 1. **ANEXOS** |
| **Menú.**    **Se agregan números al presionar 1. (se agregaron 10 cada que se presiona se agregaron 3)**    **Se imprime con en 5.**    **Se ordena con el 4.**    **Se busca el 300133.**      **Se agregan a archivo con el 6.**    **Se elimina el 322774** |

|  |
| --- |
| 1. **REFERENCIAS** |
| **JAVA SWING,**  Marc Loy, Robert Eckstein, Dave Wood, James Elliott, Brian Cole.  second edition.  O’Reilly. 2002.  **Como programar en C/C++**  H.M. Deitel/ P.J. Deitel  Segunda edición  Editorial: Prentice Hall.  ISBN:9688804711  **ESTRUCTURAS DE DATOS CON C Y C++**  YEDIYAH LANGSAM; MOSHE AUGENSTEIN  Segunda edición  Editorial: Prentice Hall.  ISBN: 9789688807989  **Ligas de Referencia:**  http://c.conclase.net/ficheros/index.php |