Archivos y su procesamiento ... Continuación

Lectura de datos provenientes de archivos de texto

Para leer datos de un archivo podemos utilizar:

a) El operador de redirección de flujo de entrada: **>>**. Se utiliza como ya sabemos hacerlo, con la diferencia que en esta ocasión ponemos a su izquierda el nombre del flujo del cuál queremos extraer la próxima cadena de texto delimitada por carácteres espacio en blanco. A la derecha puede colocarse una variable de tipo string o char [ ] (arreglo de carácteres) para capturar la entrada al programa.

b) El comando get( ). Tiene varias formas de uso, con o sin argumentos. Puede extraer un solo carácter o una cadena, de acuerdo a como se escriba el comando. En su forma más simple, lee un carácter de un archivo y retorna el entero correspondiente a ese carácter en la tabla ASCII. Utilizado con este propósito, su sintaxis es: Nombre lógico.get( ).

c) El comando getline( ). Encontramos dos versiones de este comando: como función y como método. Lo hemos venido utilizando para leer líneas desde teclado, colocando cin como primer argumento. Para extraer datos de un archivo de texto solo debemos colocar, en el primer argumento, el nombre del flujo que representa al archivo en nuestro programa. Recordemos su sintaxis: getline(Nombre lógico, Variable, Carácter delimitador).

d) El comando read( ). Lee un conjunto de carácteres del archivo de entrada. Su sintaxis es: Nombre Lógico.read(Variable, Tamaño). Donde Variable es la variable en la que se colocará la secuencia de caracteres proveniente del archivo. El Tamaño es la longitud de esta cadena, que puede ser calculado con sizeof( ) o de cualquier otra forma, debe ser un número entero. Este comando se usa preferentemente para leer contenido en archivos binarios, haciendo la conversión de tipo apropiada del primer argumento. Lo veremos luego.

Por ejemplo:

1) Escriba un programa que lea el primer carácter de un archivo y lo muestre en pantalla. Luego lea un segundo carácter y muestre su valor en la tabla ASCII.

Para efectos de este programa se creó el archivo archivo07.txt con Bloc de Notas y se escribió en él la frase Me gustan las pupusas.

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    unsigned char car;
    unsigned int x;

    cout << endl;
    cout << "LEER DATOS DESDE UN ARCHIVO" << endl << endl;

    ifstream archivo;
    archivo.open("archivo07.txt");
```

```
    car = archivo.get( );
    x = archivo.get( );

    cout << "El carácter leído es: " << car << endl;
    cout << "El siguiente carácter tiene el valor: " << x;
    cout << " en la tabla ASCII" << endl;

    archivo.close( );

    cout << endl;
    return 0;
}
```

La salida de este programa es:

```
LEER DATOS DESDE UN ARCHIVO

El carácter leído es: M
El siguiente carácter tiene el valor: 101 en la tabla ASCII
```

(Verifique en la tabla ASCII que la posición asignada a la letra **e** es la **101**)

2) Escriba un programa que lea todas las líneas de un archivo de texto y las almacene en un arreglo. Luego despliegue el contenido del arreglo.

Para efectos de este programa se creó el archivo archivo08.txt con Bloc de Notas y se escribió en él lo siguiente:

```
El mono Pepito
y el oso Pipón
visitan la case
del gnomo Pompom.
```

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    string arr[25];
    int i, k;

    cout << endl;
    cout << "LEER FRASES DESDE UN ARCHIVO" << endl << endl;

    ifstream archivo;
    archivo.open("archivo08.txt");

    i = 0;
    while(!archivo.eof( ))
        getline(archivo, arr[i++], '\n');

    cout << "Las frases del archivos son:" << endl;
    for(k = 0; k < i; k++)
        cout << arr[k] << endl;

    archivo.close( );

    cout << endl;
    return 0;
```

```
}
```

Note en la condición del lazo de lectura la utilización del método **eof( )** para detectar si se ha llegado al final del archivo. Note también que el primer argumento de la función **getline** es el nombre lógico del archivo, es decir, el nombre del flujo que lo representa.

3) Escriba un programa que lea todo el contenido de un archivo, palabra por palabra. Despliegue el contenido en pantalla a medida que lo va leyendo, para verificar los elementos leídos.

Para efectos de este programa se hace uso del archivo archivo08.txt que se creó para demostrar el programa anterior.

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    string dato;
    int i, k;

    cout << endl;
    cout << "LEER PALABRA POR PALABRA DESDE UN ARCHIVO" << endl << endl;

    ifstream archivo;
    archivo.open("C:/Users/UCA/Documents/ProyectosVSCode/archivo08/archivo08.txt");

    i = 0;
    while(archivo >> dato)
        cout   << dato << endl;

    archivo.close( );

    cout << endl;
    return 0;
}
```

La salida de este programa es:

```
LEER PALABRA POR PALABRA DESDE UN ARCHIVO

El
mono
Pepito
y
el
oso
Pipón
visitan
la
casa
del
gnomo
Pompom.
```

Notemos que la lectura simple con el operador de redirección de flujo de entrada, >>, lee nada más los datos separados por los carácteres espacio en blanco. Será importante considerar cuando

necesitamos recuperar líneas completas y cuando necesitamos recuperar los diferentes elementos por separado.

En este caso, la condición del lazo se pudo haber reescrito así: **while(archivo >> dato)**. De esta forma, el lazo completo hubiera sido:

```
while(archivo >> dato)
    cout   << dato << endl;
```

Otra cosa que debemos notar es que, en esta ocasión, se ha utilizado un archivo que se encuentra en una carpeta distinta. Para localizarlo se ha utilizado la ruta absoluta (desde la raíz del disco). Observe que para separar cada carpeta de la ruta se ha utilizado el carácter /. Se podría haber utilizado también la plica invertida, pero dado que es un carácter de control, se debería utilizar doble plica invertida cada vez que se requiera, \\, así: **C:\\Users\\UCA\\Documents\\ProyectosVSCode\\archivo08\\archivo08.txt**

4) Escriba un programa que lea los datos sobre artículos comprados en un almacén.

Para efectos de este programa se creó el archivo archivo10.txt con Bloc de Notas y se escribió en él lo siguiente:

```
Tornillos 7 0.15
Tuercas 5 0.15
Pupusas 12 0.50
Empaques 4 0.25
Pastelitos 6 0.25
Jabonera 1 2.50
```

```
#include <iostream>
#include <fstream>

using namespace std;

int main() {
    string nombre;
    int cantidad;
    float precio;
    int i, k;

    cout << endl;
    cout << "LEER PALABRA POR PALABRA DESDE UN ARCHIVO" << endl << endl;

    ifstream archivo;
    archivo.open("archivo10.txt");

    while(archivo >> nombre){
        archivo >> cantidad;
        archivo >> precio;
        cout   << "Se compraron " << cantidad << " unidades de ";
        cout << nombre << " a $" << precio << endl;
    }

    archivo.close( );

    cout << endl;
    return 0;
}
```

La salida de este programa es la siguiente:

LEER PALABRA POR PALABRA DESDE UN ARCHIVO

Se compraron 7 unidades de Tornillos a $0.15
Se compraron 5 unidades de Tuercas a $0.15
Se compraron 12 unidades de Pupusas a $0.5
Se compraron 4 unidades de Empaques a $0.25
Se compraron 6 unidades de Pastelitos a $0.25
Se compraron 1 unidades de Jabonera a $2.5

En este caso se ha hecho un ejemplo sencillo solo para demostrar que la lectura con el operador >> es sensible al tipo de la variable que recibe la entrada. Como puede verse, por cada fila hay una cadena, un entero y un real. Este operador, y el operador de redirección de flujo de salida, operan tal y como lo hacen cuando la entrada/salida es por consola. Pero cuando los utilizábamos en nuestros primeros programas no nos cuestionábamos la diferencia entre el manejo de los distintos tipos de datos. Los separadores entre un dato y otro son siempre los carácteres espacio, cambio de línea o tabulación.

Como puede verse, los datos en el archivo de texto no tienen formato, en el sentido de que no están alineados por columnas. Pero sí contienen, al menos, un espacio como separador de cada dato. Para construir un archivo de texto con formato más legible deberíamos alinear en forma columnar las cantidades y los nombres. Los nombres deberían estar justificados a la izquierda, los enteros deberían estar justificados a la derecha y las cantidades monetarias deberían estar aproximadas a dos decimales y alineadas por el punto decimal. Tanto la entrada como la salida puede formatearse, utilizando comandos de la librería iomanip de C++, para usarse en combinación con cin y cout. Hay otros comandos más antiguos, aun disponibles, como printf( ), fprintf( ), sprintf( ), scanf( ), fscanf( ), que nos ayudan a controlar, con mucho detalle, la entrada y salida con formato. Recordar que sprintf( ) lo utilizamos para convertir una cadena a real con dos decimales, hace unas clases. Es decir que sí podemos darle formato a las cosas que manipulamos para entrada y para salida.

La necesidad de delimitar las cantidades en los archivos es importante, para saber dónde comienza un dato y dónde termina, sobre todo cuando datos de texto están involucrados. Está claro que si el nombre de un artículo tuviera más de una palabra y además las longitudes de esas frases fueran diferentes, eso haría complicada la lectura. Por eso es necesario utilizar formatos bastante bien definidos cuando se trabaja con archivos de texto que lo requieran. Una salida a ello es manejar estructuras en los archivos, en lugar de texto ASCII, dado que las estructuras las podemos usar como patrón o plantilla del conjunto de datos.

Ya que hemos resuelto un problema en el cual se maneja un archivo de texto en el que se representan diferentes tipos de datos, es conveniente pasar a estudiar los archivos binarios. En concreto, los que almacenan estructuras con campos de diverso tipo.

Los archivos en modo binario los veremos en otro documento.