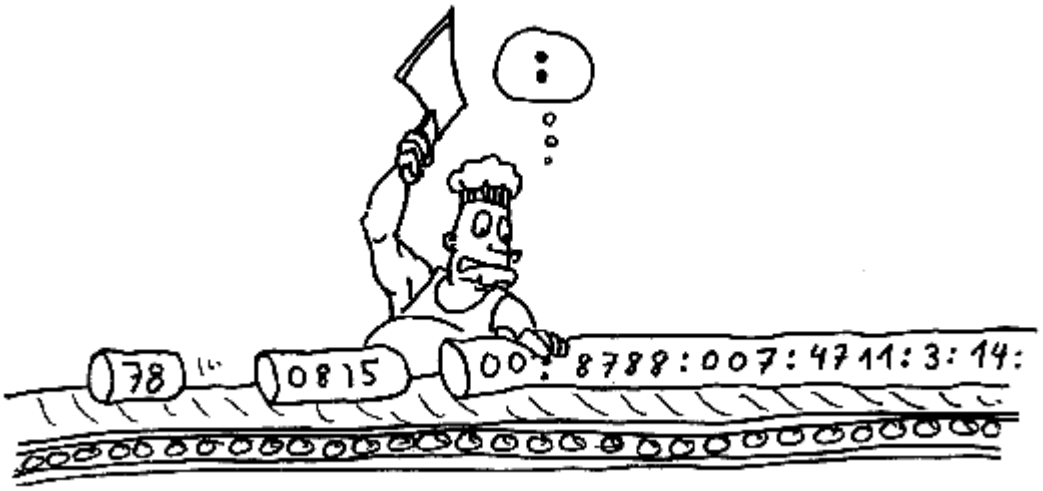
# Práctica 1: Tokenizador




Autor: Antonio Martínez Santa
Universidad de Alicante

## Contexto y problema a solucionar

Se pide construir la clase Tokenizador que segmenta strings en palabras, con la opción de detectar una serie de casos especiales de palabras. Asimismo, hay que calcular la complejidad TEÓRICA temporal y espacial del algoritmo de tokenización de casos especiales tal y como se ha visto en clase. Durante esta práctica se deberá implementar un tokenizador con unos casos especiales en concreto que se tratan más adelante en este mismo documento. La implementación será en el lenguaje de programación C++ y será nuestra propia elección la que optemos por optimizar el código que nos da el profesor en el enunciado de la práctica o hacer por nuestra propia cuenta un tokenizador optimizado.

## Método Tokenizador()

Este método será el constructor por defecto de la clase Tokenizador, éste deberá cumplir una especificación que se nos requiere.

### Implementación del método

```
Tokenizador::Tokenizador(){
    delimiters = ",;:.-/+*\\ '\"{}[]()<>¡!¿?&#=\t@";
    for(int i = 0; i < 256; i++){
        delimitadores[i] = false;
    }
    for(auto delimitador : delimiters){
        delimitadores[static_cast<unsigned char> (delimitador)] = true;
    }
    casosEspeciales = true;
    pasarAminuscSinAcentos = false;
}
```

### Mejoras aplicadas

Pero por experiencia de otra asignatura llamada Análisis y Diseño de Algoritmos voy a añadirle una pequeña modificación que hará que sea más eficiente. *¿Por qué?* Implementando un **array de booleanos con 256 casillas** podemos obtener una eficiencia a la hora de recorrer strings ya que esto no sería necesario, con comprobar si es un delimitador en este array de booleanos (y no sería necesario recorrerlo entero si no con su valor en ISO-8859-1 en el array es TRUE sabemos que sí lo es) podemos saber si realmente lo es sin tener que recorrer el string con **O(K) siendo K el número de elementos**. Estaríamos rebajando esta complejidad a **O(1) en consulta** y solo ocuparía **256 bytes de memoria** (1 por cada posible carácter).

```
Tokenizador::Tokenizador(){
    delimiters = ",;:.-/+*\\ '\"{}[]()<>¡!¿?&#=\t@";
    for(int i = 0; i < 256; i++){
        delimitadores[i] = false;
    }
    casosEspeciales = true;
    pasarAminuscSinAcentos = false;
}
```

Y luego vamos rellenando por cada uno de los delimitadores que hay por defecto en el constructor vamos modificando las distintas posiciones en el array de booleanos con

```
for(auto delimitador : delimiters){
    delimitadores[static_cast<unsigned char> (delimitador)] = true;
}
```

Obteniendo una concatenación de **O(256) + O(n) = O(n)** como complejidad final siendo n el número de caracteres al inicializar.

### Posibles contras

Esto podría llegar a suponer un desgaste de memoria si no vamos a llegar a tener valores cercanos a 256 delimitadores ya que para 2 delimitadores usados habría 254 bytes que no estarían siendo usados por la memoria estática que ya hemos reservado pero para este caso me conformo.

## Método Tokenizador(const string&, const bool&, const bool&)

Este método es el constructor parametrizado, éste deberá crear una instancia de Tokenizador con los valores que se pasen por parámetro de manera correcta.

### Implementación del método

```
Tokenizador::Tokenizador(const string& delimitadoresPalabra, const bool& kcasosEspeciales, const bool& minuscSinAcentos){
    delimiters = "";
    for(int i = 0; i < 256; i++){   // inicializacion previa
        delimitadores[i] = false;
    }
    for(int i = 0; i < delimitadoresPalabra.size(); i++){
        auto caracter = delimitadoresPalabra[i];
        if(delimitadores[static_cast<unsigned char>(caracter)] == false){   // cambiamos a unsigned para no tener indices
            delimiters += caracter;
            delimitadores[static_cast<unsigned char>(caracter)] = true;     // delimitador introducido ya, evita repes
        }
    }
    casosEspeciales = kcasosEspeciales;
    pasarAminuscSinAcentos = minuscSinAcentos;
}
```

### Mejoras aplicadas

No hay una mejora significativa con respecto al método anterior, simplemente se ha seguido con la metodología de aplicar el array de booleanos donde además también se comprueba que no se introduzcan caracteres repetidos (cumple la especificación). El uso de *static_cast* es para asegurar que se accede de manera segura al array de booleanos. Concluimos con una complejidad **O(n)** al tener 2 for SIN anidar siendo **O(256) + O(n) = O(n)** siendo **n** el número de delimitadores en el parámetro.

Comprobación de funcionamiento

Para ello vamos a introducir caracteres repetidos en la inicialización de un Tokenizador.

```
int main() {
    Tokenizador tokenizador(",:..:", true, true);
    cout << "Delimitadores configurados: \""
    << tokenizador.DelimitadoresPalabra() << "\"" << endl;
    return 0;
}
```

Así como vemos hemos introducido caracteres repetidos que son ':' y '.'. La salida ha sido la siguiente:

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./main
Delimitadores configurados: ",:."
```

Comprobamos que realmente funciona correctamente.

## Método Tokenizador(const Tokenizador&)

Este método es el constructor de copia, éste se encargará de copiar el contenido de un tokenizador en otro nuevo con el mismo contenido.

Implementación del método

```
Tokenizador::Tokenizador(const Tokenizador& t){
    delimiters = t.delimiters;
    for(int i = 0; i < 256; i++){
        delimitadores[i] = t.delimitadores[i];  // cop
    }
    casosEspeciales = t.casosEspeciales;
    pasarAminuscSinAcentos = t.pasarAminuscSinAcentos;
}
```

Mejoras aplicadas

Otra vez como mejora aplicada seguimos con la implementación del array de booleanos. En este caso en específico como el objeto por parámetro ya es un Tokenizador tendrá su array de 256 bytes o posiciones definido por lo que su complejidad sería **O(n)** ya que lo más costoso es la operación de copiar el string de delimitadores, el resto sería **O(1)** todos (menos el for que es **O(256) ≈ O(1)**).

## Método ~Tokenizador()

Este método es el encargado de poner el valor de delimiters a string vacío para proceder a la eliminación del objeto una vez ya no vaya a ser usado. Se eliminará con delete el objeto que se desee.

Implementación del método

```
Tokenizador::~Tokenizador(){
    delimiters = "";
}
```

Comprobación de funcionamiento

Ponemos a prueba su funcionamiento con este pequeño main.

```
int main() {
    Tokenizador* tokenizador = new Tokenizador(";.:", false, true);
    cout << "Delimitadores: " << tokenizador->DelimitadoresPalabra() << endl;
    delete tokenizador;

    return 0;
}
```

Y esta es la salida que obtenemos.

```
==458996== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright
==458996== Command: ./main
==458996==
Delimitadores: ;.:
==458996==
==458996== HEAP SUMMARY:
==458996==     in use at exit: 0 bytes in 0 blocks
==458996==   total heap usage: 3 allocs, 3 frees, 75,048 bytes allocated
==458996==
==458996== All heap blocks were freed -- no leaks are possible
==458996==
==458996== For lists of detected and suppressed errors, rerun with: -s
==458996== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## Método Tokenizar(const string&, list<string>&)

Este método es el encargado principal de **tokenizar un string** y **meterlo en la lista de string** que serán los **tokens** sacados de ese método. En un principio propuse una implementación de dos bucles anidados pero me obsesioné con la eficiencia por lo que decidí obviar esa solución y decidí pensar en uno un poquito más elaborado.

## Implementación del método

```cpp
void Tokenizador::Tokenizar(const std::string& str, std::list<std::string>& tokens) const{
    string palabra;         // para almacenar cada palabra
    char caracter;

    for(int i = 0; i < str.length(); i++){
        caracter = str[i];
        if(delimitadores[static_cast<unsigned char>(caracter)] == true){    // si es un del
            if(palabra.length() != 0){
                tokens.push_back(palabra);
                palabra.clear();
            }
        }else{
            palabra += caracter;
        }
    }

    // si la ultima no está vacia la añadimos
    if(palabra.length() != 0){
        tokens.push_back(palabra);
    }
}
```

## Mejoras aplicadas

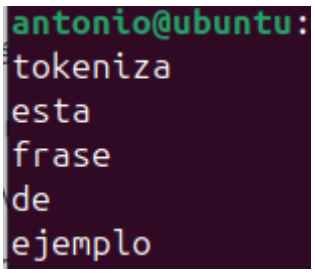
Al hacer uso de solo 1 bucle for tendremos una complejidad **O(n)** también en su caso peor que será la longitud completa del string. Después de haber estado investigando sobre si el uso de string.clear() sería el adecuado he analizado que casi todos los tokens van a ser **tokens cortos**, muy **rara vez** nos vamos a encontrar tokens con **palabras exageradamente largas** por lo que string.clear() será de **complejidad O(1)** en la mayoría de casos al hacer uso de **SSO** (Small String Optimization). Concluimos que la suma de todas las complejidades **O(n) seguidas** + **O(1)** de accesos a memoria rápidos además por la implementación del array de booleanos es lo más óptimo que se me ha ocurrido.

## Comprobación de funcionamiento

Le asigno este pequeño main para comprobar que esto funciona realmente.

```cpp
int main() {
    string frase = "tokeniza esta frase de ejemplo";
    list<string> tokens;
    Tokenizador tokenizador(" ,:", false, true);

    tokenizador.Tokenizar(frase, tokens);
    for(auto f : tokens){
        cout << f << endl;
    }

    return 0;
}
```

Y esta es la salida que he obtenido.




## Método Tokenizar(const string&, const string&)

Este método consiste en tokenizar el contenido de un fichero en otro de salida con un token por línea, es decir, cada token separado por un salto de línea.

### Implementación del método

```
bool Tokenizador::Tokenizar(const string& i, const string& f) const{
    ifstream entrada(i);
    ofstream salida(f);
    string palabra = "";
    char caracter;

    if(entrada.is_open() == false){
        cerr << "El fichero no se ha podido abrir o no existe\n";
        return false;
    }else{
        while(entrada.get(caracter)){
            if(delimitadores[static_cast<unsigned char>(caracter)] == true){
                if(palabra.length() != 0){
                    salida << palabra;       // printea la palabra
                    palabra.clear();        // limpiamos para la siguiente
                    salida << "\n";
                }
            }else{
                palabra += caracter;     // concatenamos la palabra
            }
        }

        if(palabra.size() != 0){     // si la última no está vacía
            salida << palabra;
            palabra.clear();
        }

        entrada.close();
        salida.close();

        return true;
    }
```

### Mejoras aplicadas

Primero hice la implementación de arriba y luego decidí modificarla ya que me dí cuenta que estaba haciendo literalmente lo mismo que en el primer método si le pasaba la frase entera

y no carácter a carácter. Se simplificó mucho más la lógica implementada. La nueva es la siguiente:

```
bool Tokenizador::Tokenizar(const string& i, const string& f) const{
    ifstream entrada(i);
    string linea;
    list<string> tokens;

    if(entrada.is_open() == false){
        cerr << "ERROR: No existe el archivo: " << i;
        return false;
    }else{
        while(entrada.eof() == false){
            linea = "";
            getline(entrada, linea);
            if(linea.length() != 0){
                this->Tokenizar(linea, tokens);
            }
        }
        entrada.close();
        ofstream salida(f);
        list<string>::iterator itS;
        for(itS = tokens.begin(); itS != tokens.end(); itS++){
            salida << (*itS) << "\n";
        }
        salida.close();
        return true;
    }
}
```

Es una implementación muy similar a la del fichero de la práctica pero con algunas implementaciones que mejoran el rendimiento. El uso de ***"\n"*** para ahorrar tiempo de ejecución, otra de las experiencias que me llevé de ADA y el uso del método implementado anteriormente que cuenta con avances para hacerlo óptimo con complejidad **O(n)**. Como el bucle se **ejecutará según las líneas** ("L") que tenga el fichero podemos decir que tendrá una **complejidad O(n * L)** final.

## Comprobación de funcionamiento

Preparé un pequeño main para ver cómo se comportaba el código, el main es el siguiente:

```
int main() {
    string entrada = "entrada.txt";
    string salida = "salida.txt";
    Tokenizador t;
    t.Tokenizar(entrada, salida);

    return 0;
}
```

El contenido del fichero de entrada es la siguiente frase “esto es una prueba para tokenizar ficheros“.
Al ejecutar este main obtenemos la siguiente salida.

salida.txt
```
esto
es
una
prueba
para
tokenizar
ficheros
```

## Método Tokenizar(const string&)

Este método hace el mismo tratamiento que el anterior pero modificando el nombre del archivo de salida es el nombre de entrada con extensión .tk.

### Implementación del método

```
bool Tokenizador::Tokenizar(const string & i) const{
    bool correcto = false;
    string nombreSalida = i + ".tk";
    if(this->Tokenizar(i, nombreSalida)){
        correcto = true;
    }
    return correcto;
}
```

### Mejoras aplicadas

Como mejora se ha aplicado la reutilización de código en el método que hicimos anteriormente ya que hace exactamente lo mismo, solo cambia el nombre del fichero de salida. Al ser una declaración la complejidad del método sigue siendo la misma que en el caso anterior, es decir, **complejidad O(n * L)**.

### Comprobación de funcionamiento

He definido el siguiente main para comprobar el funcionamiento, es realmente el mismo que el anterior pero sin fichero de salida como tal. El contenido del fichero “entrada.txt” es el mismo que en el anterior ejemplo.

```
int main() {
    string entrada = "entrada.txt";
    Tokenizador t;
    t.Tokenizar(entrada);

    return 0;
}
```

Esta es la salida obtenida.

entrada.txt.tk

```
esto
es
una
prueba
para
tokenizar
ficheros
```

Comprobamos su correcto funcionamiento.

## Método TokenizarListaFicheros(const string&)

Este método será el encargado de tokenizar, es decir, aplicar el mismo proceso que llevamos haciendo en todos los métodos a cada uno de los ficheros contenidos dentro del fichero que se pasa por parámetro. Se realizará la tokenización de cada uno de los ficheros y se devolverá true si se ha realizado correctamente, en caso contrario se enviará un mensaje de error con el mensaje correspondiente.

### Implementación del método

```
bool Tokenizador::TokenizarListaFicheros(const string& i) const{
    bool correcto = true;
    ifstream ficheros(i);
    string ficheroATokenizar;
    struct stat dir;
    list<string> tokens;
    string lineaEnFichero = "";

    if(ficheros.is_open() == true && stat(i.c_str(), &dir) != 0 || S_ISDIR(dir.st_mode) == false){
        while(getline(ficheros, ficheroATokenizar)){    // cogemos linea a linea
            tokens.clear();
            if(stat(ficheroATokenizar.c_str(), &dir) != 0 || S_ISDIR(dir.st_mode) == false){ // comprobacion directorio
                ifstream fichero(ficheroATokenizar);
                if(fichero.is_open() == true){
                    while(getline(fichero, lineaEnFichero)){
                        Tokenizar(lineaEnFichero, tokens);
                    }
                    ofstream salida(ficheroATokenizar + ".tk");
                    for(auto token : tokens){
                        salida << token << "\n";
                    }
                    fichero.close();
                    salida.close();
                }else{
                    cerr << "ERROR: No existe el archivo " << ficheroATokenizar << " pero no se interrumpe la ejecución" << endl;
                    continue;
                }
            }else{
                cerr << "ERROR: El archivo " << ficheroATokenizar << " es un directorio pero no se interrumpe la ejecución" << endl;
                continue;
            }
        }
    }else{
        cerr << "ERROR: El archivo " << i << " no existe o se trata de un directorio" << endl;
        return false;
    }
    return correcto;
}
```

### Mejoras aplicadas

Este es el primer prototipo de método que se me ha ocurrido cumpliendo con la especificación que consta sobre el mismo. Como este método será el utilizado para hacer las pruebas de velocidad y memoria éste es el que hay que optimizar más en profundidad además de que el resto de métodos mencionados anteriormente no tenían una complejidad muy elevada de implementación al contrario que este que tiene más chicha. Al ser una

implementación precaria puede que no sea la óptima por lo que vamos a intentar optimizar el rendimiento de su complejidad temporal y su complejidad espacial. En primer lugar voy a generar un contenido similar en bytes al que se usará para la corrección, es decir, 19.900KB repartidos en 10 archivos porque no iba a ponerme a generar 717 archivos pero vamos que el funcionamiento es el mismo. Para ello usé un programa en Python para generar archivos con palabras aleatorias con líneas de longitud entre 5 y 15 palabras por línea.

fileGenerator.py > ...

```python
import random
import os

output_dir = "token_files"
os.makedirs(output_dir, exist_ok=True)

num_files = 10
file_size_kb = 2000
file_size_bytes = file_size_kb * 1024

words = ["token", "tokenizar", "test", "prueba", "probando", "aleatorio"]

min_words_per_line = 5
max_words_per_line = 15

for i in range(1, num_files + 1):
    file_path = os.path.join(output_dir, f"tokens_{i}.txt")

    with open(file_path, "w") as f:
        current_size = 0
        while current_size < file_size_bytes:
            num_words = random.randint(min_words_per_line, max_words_per_line)
            line = " ".join(random.choices(words, k=num_words)) + "\n"
            f.write(line)
            current_size += len(line)
```

Ahora ya podemos empezar a usar la herramienta memory.cpp que se nos proporciona. Esta herramienta mide (por lo que he visto) el pico de memoria más alto en el programa por lo que por mucho que vayas utilizando menos memoria se quedará con el más alto que consiga tener. Esta es la estructura de uno de los ficheros:

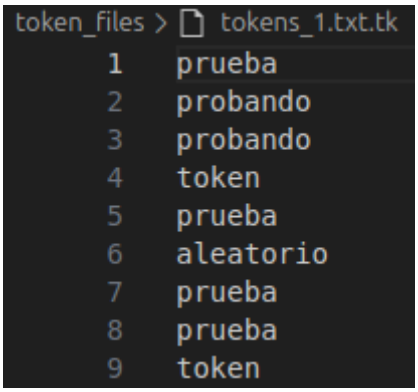
token_files > tokens_1.txt

```
prueba probando probando token prueba
aleatorio prueba prueba token aleatorio aleatorio aleatorio tokenizar probando prueba
aleatorio aleatorio tokenizar tokenizar tokenizar probando tokenizar probando prueba probando prueba test tokenizar
prueba aleatorio prueba tokenizar token aleatorio tokenizar aleatorio tokenizar prueba
token tokenizar token tokenizar token token test tokenizar prueba
test tokenizar prueba prueba tokenizar test test aleatorio aleatorio token
aleatorio prueba aleatorio tokenizar probando probando tokenizar test probando token test tokenizar prueba aleatorio
test token aleatorio token tokenizar prueba tokenizar token tokenizar test tokenizar
aleatorio token tokenizar aleatorio aleatorio tokenizar tokenizar aleatorio token test prueba test
tokenizar test test prueba tokenizar token probando token prueba
tokenizar test aleatorio prueba test tokenizar test
token token probando test token aleatorio aleatorio tokenizar test probando
test aleatorio prueba probando test probando aleatorio aleatorio prueba prueba token token token
probando aleatorio token tokenizar test prueba tokenizar
token token probando tokenizar token test tokenizar test probando probando test tokenizar test test probando
token tokenizar tokenizar aleatorio aleatorio probando aleatorio probando test aleatorio
prueba tokenizar prueba token aleatorio test tokenizar prueba probando probando prueba probando probando tokenizar
aleatorio aleatorio probando prueba test aleatorio test test
probando tokenizar probando token probando prueba test
probando tokenizar aleatorio probando tokenizar test test test test test token aleatorio tokenizar
prueba token token test aleatorio aleatorio prueba test tokenizar prueba
token test probando test aleatorio prueba token
token token tokenizar tokenizar token token test
test token test prueba tokenizar probando probando
probando tokenizar token test tokenizar aleatorio test prueba prueba tokenizar prueba prueba token prueba
```

Con esto ya podemos empezar a probar qué tal va el programa. Compilé el proyecto y el memory y ahora sí este era el resultado que obtenía.

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./memory ./main
Tiempo de ejecución: 0.725188 s

Memoria total usada: 18220 Kbytes
  "   de datos: 18088 Kbytes
  "    de pila: 132 Kbytes
```

Un tiempo de ejecución esperado la verdad, no creo que se salga de algo normal con las optimizaciones que le implementé (mirando los tiempos de años anteriores de la transparencia de teoría parece bastante aceptable), el problema viene con la memoria, ocupa muchísima memoria (superando al máximo de memoria de las transparencias por muchísimo) por lo que esto hay que optimizarlo. Ah, por cierto, sí, el resultado de la tokenización fue correcto, aquí la prueba:




Esto fue realmente sencillo de solucionar, fue problema mío que al verlo a simple vista pues no supe verlo pero luego me di cuenta de que al almacenar los tokens **ABSOLUTAMENTE TODOS** en una lista de tokens ***list<string> tokens*** y no borrarla por cada string que tokenizabas pues claro era una locura la cantidad de memoria que se está usando. ¿Solución? Fácil, tokenizar, escribir en fichero y borrar tokens. Con esto te estás cargando el tener que almacenar todos los tokens en memoria local que para qué los quieres y ahora asumiendo que van a ser frases cortas o con saltos de línea frecuentes (o no necesariamente) no tendrás que almacenar todo el contenido de todos los tokens. Los cambios significativos fueron estos marcados aquí:

```cpp
bool Tokenizador::TokenizarListaFicheros(const string& i) const{
    bool correcto = true;
    ifstream ficheros(i);
    string ficheroATokenizar;
    struct stat dir;
    list<string> tokens;
    string lineaEnFichero = "";

    if(ficheros.is_open() == true && stat(i.c_str(), &dir) != 0 || S_ISDIR(dir.st_mode) == false){
        while(getline(ficheros, ficheroATokenizar)){    // cogemos linea a linea
            tokens.clear();
            if(stat(ficheroATokenizar.c_str(), &dir) != 0 || S_ISDIR(dir.st_mode) == false){ // comprobacion directorio
                ifstream fichero(ficheroATokenizar);
                ofstream salida(ficheroATokenizar + ".tk");
                if(fichero.is_open() == true){
                    while(getline(fichero, lineaEnFichero)){
                        tokens.clear();
                        Tokenizar(lineaEnFichero, tokens);
                        for(auto token : tokens){
                            salida << token << "\n";
                        }
                    }
                    fichero.close();
                    salida.close();
                }else{
                    cerr << "ERROR: No existe el archivo " << ficheroATokenizar << " pero no se interrumpe la ejecución" << endl;
                    continue;
                }
            }else{
                cerr << "ERROR: El archivo " << ficheroATokenizar << " es un directorio pero no se interrumpe la ejecución" << endl;
                continue;
            }
        }
    }else{
        cerr << "ERROR: El archivo " << i << " no existe o se trata de un directorio" << endl;
        return false;
    }
    return correcto;
```

Limpiar los tokens en cada línea

Escribir directamente en el fichero de salida

Con estos cambios se consiguen estas marcas.

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./memory ./main
Tiempo de ejecución: 0.641087 s

Memoria total usada: 400 Kbytes
  "    de datos: 268 Kbytes
  "     de pila: 132 Kbytes
```

Bastante mejor en este aspecto y el tiempo bastante aceptable, una buena optimización. Por último queda comprobar que ha hecho su trabajo. Este es el resultado de la tokenización.

token_files > tokens_1.txt.tk

```
1  prueba
2  probando
3  probando
4  token
5  prueba
6  aleatorio
7  prueba
8  prueba
9  token
```

token_files > tokens_2.txt.tk

```
1  tokenizar
2  prueba
3  prueba
4  test
5  tokenizar
6  prueba
7  test
8  tokenizar
9  aleatorio
```

token_files > tokens_3.txt.tk

```
1  tokenizar
2  tokenizar
3  probando
4  prueba
5  tokenizar
6  token
7  token
8  tokenizar
9  prueba
```

Como se puede observar siguen patrones distintos porque se generan aleatoriamente las palabras de cada frase. Todo esto sin caracteres especiales, aún no llegamos a ellos. Me sorprendió bastante el rendimiento que tenía, no parecía ir nada mal en cuanto a implementación. Cambié una pequeña cosa en el *for poniendo (const auto&)* al escribir para evitar copias de caracteres al recorrer el array.

Tuve que añadir una pequeña modificación para adaptar a la especificación sobre si había algún fichero que no estaba de manera correcta (era un directorio o no existía) devolver false ya que si había fallos aunque se tokenizasen los ficheros de igual manera.

## Método TokenizarDirectorio(const string&)

Este método será el encargado de tokenizar un directorio completo siguiendo la especificación que se nos proporciona. He usado la implementación proporcionada por el profesor en el enunciado de la práctica ya que me parece concisa y sencilla.

### Implementación del método

```
bool Tokenizador::TokenizarDirectorio(const string& i) const{
    bool correcto = true;
    struct stat dir;
    int error = stat(i.c_str(), &dir);
    if(error != -1 && S_ISDIR(dir.st_mode) == true){
        string cmd = "find " + i + " -follow |sort > .lista_fich";
        system(cmd.c_str());
        return TokenizarListaFicheros(".lista_fich");
    }else{
        cerr << "ERROR: La ruta " << i << " no es un directorio o los ficheros no se han podido tokenizar" << endl;
        return false;
    }
}
```

### Mejoras aplicadas

Como pone en el enunciado de la práctica se sugiere implementar mejoras como la detección de directorios en TokenizarListaFicheros (cosa que tengo hecha pero no sé si tiene que hacer algo por el momento o simplemente skipear los que sean directorios) ya que como se puede apreciar en el apartado anterior hay una comprobación de que sea un directorio el fichero leído o no.

### Comprobación de funcionamiento

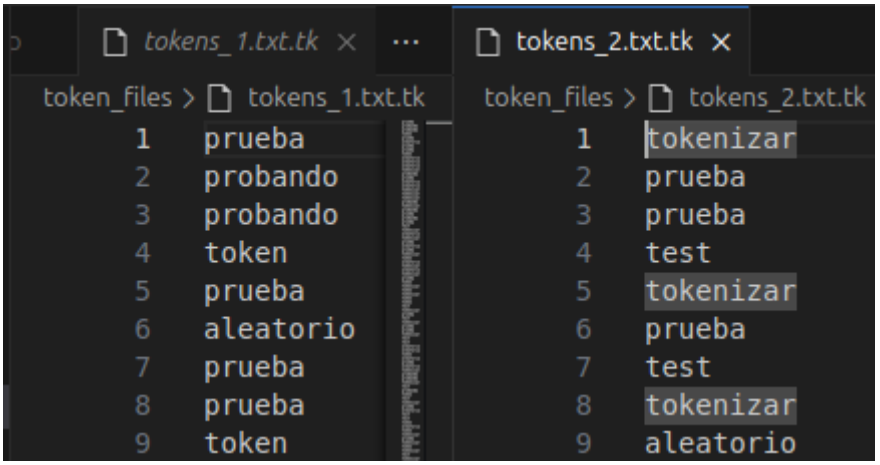
Para comprobar que funciona correctamente voy a generar un main para probarlo que es el siguiente:

```
int main() {
    Tokenizador t;
    string tokenizar = "token_files";

    auto start = high_resolution_clock::now();

    //bool resultado = t.TokenizarListaFicheros("tokenizar");
    bool resultado = t.TokenizarDirectorio(tokenizar);

    auto stop = high_resolution_clock::now();
    duration<double> duration = stop - start;
    cout << "Tiempo de ejecución: " << duration.count() << " s" << endl;
    cout << "Tokenización completada: " << (resultado ? "Sí" : "No") << endl;

    return 0;
}
```

Aquí vamos a tokenizar el directorio token_files que es el que contiene los 10 archivos con aproximadamente **2000 * 1024 bytes** haciendo así **10 * 2000 * 1024 bytes** de tamaño total. Esta es la salida que se obtuvo de realizar la tokenización:

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./memory ./main
ERROR: El archivo token_files es un directorio pero no se interrumpe la ejecucion
Tiempo de ejecución: 0.605167 s
Tokenización completada: No

Memoria total usada: 400 Kbytes
  "   de datos: 268 Kbytes
  "    de pila: 132 Kbytes
```

A pesar de ser un directorio el fichero pasado por parámetro sale el mensaje de error como que es un directorio pero sigue con la tokenización (lo pone en la especificación de TokenizarListaFicheros por ello sale tokenización completada: No) y por consiguiente este es el resultado:




Coincide con lo que obtuvimos usando TokenizarListaFicheros.

## Método DelimitadoresPalabra(const string&)

Este método se encarga de sustituir los delimitadores que había anteriormente por unos nuevos especificados por parámetro.

### Implementación del método

```
void Tokenizador::DelimitadoresPalabra(const string& nuevoDelimiters){
    delimiters = nuevoDelimiters;
    for(int i = 0; i < 256; i++){   // reiniciar delimitadores
        delimitadores[i] = false;
    }
    for(const auto& caracter : delimiters){ // asignar los nuevos
        delimitadores[static_cast<unsigned char>(caracter)] = true;
    }
}
```

Mejoras aplicadas

Siguiendo con la metodología del array de booleanos es de rápido acceso con una **complejidad** del método total de **O(n) + O(256) = O(n)** siendo n el número de nuevos elementos en nuevoDelimiters, *o O(d) como en clase de teoría.*

## Método AnyadirDelimitadoresPalabra(const string&)

Este método se encarga de añadir nuevos delimitadores al string de delimitadores sin repetir los caracteres que ya contiene.

Implementación del método

```
void Tokenizador::AnyadirDelimitadoresPalabra(const string& nuevoDelimiters){
    for(const auto& caracter : nuevoDelimiters){
        if(delimitadores[static_cast<unsigned char> (caracter)] == false){  //
            delimiters += caracter;
            delimitadores[static_cast<unsigned char> (caracter)] = true;
        }
    }
}
```

Mejoras aplicadas

Seguimos con la implementación de la eficiencia del array de booleanos por lo que añadir caracteres nuevos es mucho más sencillo ya que con **comprobar si ya está en ese array O(1)** podemos saber si meterlo o no.

Comprobación de funcionamiento

Para comprobarlo basta con este pequeño main que nos indica si se añadirán nuevos caracteres o no al string de delimitadores.

```
int main() {
    Tokenizador t;
    t.AnyadirDelimitadoresPalabra("Á ,;");
    cout << t.DelimitadoresPalabra() << endl;
```

Obtenemos esta salida.

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./memory ./main
,;:.-/+*\ '"{}[]()<>¿!¿?&#=     ÁÁ
```

Comprobamos así que solo se añaden los delimitadores nuevos, no los ya existentes. (En consola no se muestran correctamente los caracteres porque está en UTF-8, en los ficheros si configuras el IDE sí se ven bien).

El resto de métodos son getters y setters por lo que considero que no es necesario explicar nada de su implementación. Ahora vamos a pasar a implementar las heurísticas para la detección de palabras compuestas y casos especiales dentro de las que tendremos:

- URLs
- Números decimales
- E-mails
- Detección de acrónimos
- Multipalabras

Estos casos especiales se comprobarán en este orden por lo que empezamos con la implementación de URLs.

## Detección del caso especial: URL

Para ello según pone en la especificación al analizar las URLs debemos cumplir con una serie de pasos:

1. Solo se considerará una URL si empieza por http:, https: o ftp: (en minúsculas)
2. La URL será tratada como 1 token da igual los delimitadores que haya dentro de ella
3. Si pasarAminuscSinAcentos está a true hay que convertir Http a http (o cualquier otro por el que comience la URL) antes de evaluar la URL.
4. La URL finalizará cuando se detecte otro delimitador que no sea "_:/.?&-=#@" o un espacio en blanco.

Comenzamos con la implementación en el método Tokenizar(const string&, list<string>&) porque este es el método usado para la tokenización en el ámbito de TokenizarListaFicheros. Implementé una posible manera de cómo podría ser un prototipo de forma de analizar estas URL con este código:

```
void Tokenizador::TokenizarCasosEspeciales(const string& str, list<string>& tokens) const{
    string palabra;
    char caracter;
    int longitudString = str.length();
    bool esURL = false;

    for(int i = 0; i < longitudString; i++){
        if(i + 6 <= longitudString){
            string prefijo = str.substr(i, 6);  // contendrá https: http: ftp: en tokens de mas de 6 caracteres (o no los contendra si son tokens que no son URL)
            if(pasarAminuscSinAcentos == true){
                for(auto& caracter : prefijo)
                    caracter = tolower(caracter);
            }
            string_view vistaHTTP = prefijo.substr(0, 5);   // ahorrar memoria
            string_view vistaFTP = prefijo.substr(0, 4);
            if(prefijo == "https:" || vistaHTTP == "http:" || vistaFTP == "ftp:"){
                esURL = true;
            }
        }
        if(esURL == true){
            int j = i;      // extraer el resto de la URL
            while(j < longitudString && (isalnum(str[i]) == true || caracteresURL.count(str[i]) == 1)){ // paramos cuando encontramos espacio o no un caracterURL
                j++;
            }
            tokens.push_back(str.substr(i, j - i)); // añadimos substring de i (inicio) hasta j - i (final - inicio)
            i = j - 1;  // saltamos todos los caracteres de la URL
            esURL = false;  // para el siguiente token leido
        }else{
            if(delimitadores[static_cast<unsigned char>(str[i])] == true){
                if(palabra.empty() == false){
                    tokens.push_back(palabra);
                    palabra.clear();
                }
            }else{
                palabra += str[i];
            }
        }
    }
    if(palabra.empty() == false){
        tokens.push_back(palabra);
    }
}
```

Spoiler: bucle infinito. Al parecer no está pensado de la mejor manera. Borrón y cuenta nueva. Planteé una nueva forma de hacer esta casuística. Escribo esto después de muchísimas horas después de haber planteado esa forma, he logrado obtener una función que tiene aproximadamente 100 líneas con la que al fin funciona esto.

```cpp
void Tokenizador::TokenizarCasosEspeciales(const string& str, list<string>& tokens) const{
    string url;
    bool esURL = false;
    string palabra;
    char caracter;
    // mismo método que con delimitadores pero para el caso de URLs
    bool caracteresValidosURL[256] = {false};
    for (char c = 'a'; c <= 'z'; c++) caracteresValidosURL[static_cast<unsigned char> (c)] = true;
    for (char c = 'A'; c <= 'Z'; c++) caracteresValidosURL[static_cast<unsigned char> (c)] = true;
    for (char c = '0'; c <= '9'; c++) caracteresValidosURL[static_cast<unsigned char> (c)] = true;
    caracteresValidosURL[static_cast<unsigned char> ('_')] = true;
    caracteresValidosURL[static_cast<unsigned char> (':')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('/')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('.')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('?')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('&')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('-')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('=')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('#')] = true;
    caracteresValidosURL[static_cast<unsigned char> ('@')] = true;

    for(int i = 0; i < str.length(); i++){
        caracter = str[i];
        if(delimitadores[static_cast<unsigned char>(caracter)] == false){
            if(palabra.empty() == true){    // inicio de token
                bool esFTP, esHTTP, esHTTPS = false;
                if(i + 8 < str.length()){       // si esta pos + 8 < length puede tener https:// y
                    string posibleHTTPS = str.substr(i, 5);
                    string posibleHTTP = str.substr(i, 4);
                    string posibleFTP = str.substr(i, 3);
                    if(pasarAminuscSinAcentos == true){
                        for(auto& caracter : posibleFTP) caracter = tolower(caracter);
                        for(auto& caracter : posibleHTTP) caracter = tolower(caracter);
                        for(auto& caracter : posibleHTTPS) caracter = tolower(caracter);
                    }
                    if(posibleFTP == "ftp"){
                        esFTP = true;
                    }else if(posibleHTTP == "http"){
                        esHTTP = true;
                    }else if (posibleHTTPS == "https"){
                        esHTTPS = true;
                    }
                }
                if(esFTP == true){
                    palabra += str.substr(i, 6); // agregamos ftp://
                    i += 5;
                    esURL = true;
                    esFTP = false;
                }else if(esHTTP == true){
                    palabra += str.substr(i, 7); // agregamos http://
                    i += 6;
                    esURL = true;
                    esHTTP = false;
```

```
                }else if(esHTTPS == true){
                    palabra += str.substr(i, 8); // agregamos https://
                    i += 7;
                    esURL = true;
                    esHTTPS = false;
                }else{
                    palabra += caracter;    // no es caso especial concatena igual
                }
                if(esURL == true){  // hay que leer toda la URL
                    while(i < str.length()){    // no es anidado, el indice avanzará
                        i++;
                        caracter = str[i];
                        if(caracteresValidosURL[static_cast<unsigned char>(caracter)] == true){
                            palabra += caracter;
                        }else{
                            break;  // delimitador no válido
                        }
                    }
                    tokens.push_back(palabra);
                    palabra.clear();
                    esURL = false;
                    i--;
                    continue;
                }
            }else{                      // no es inicio de token
                palabra += caracter;    // concatenar caracteres
            }
        }else{
            if(palabra.empty() == false){   // delimitador encontrado
                tokens.push_back(palabra);
                palabra.clear();
            }
        }
    }

    if(palabra.empty() == false){   // si antes de terminar queda una palabra con contenido se agrega
        tokens.push_back(palabra);
        palabra.clear();
    }
}
```

Vamos a ponerlo a prueba con un ejemplo del enunciado. Este es el main que he generado:

```
int main() {
    Tokenizador tokenizador;
    list<string> tokens;
    string s1 = "p0 http://intime.dlsi.ua.es:8080/dossierct/index.jsp?lang=es&status=probable&date=22-01-2013&newspaper=catedraTelefonicaUA@iuii.ua.es p1 p2";

    tokenizador.Tokenizar(s1, tokens);

    cout << "Tokens obtenidos: " << endl;
    for (const auto& token : tokens) {
        std::cout << token << std::endl;
    }
    return 0;
```

Esta es la salida obtenida.

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./main
Tokens obtenidos:
p0
http://intime.dlsi.ua.es:8080/dossierct/index.jsp?lang=es&status=probable&date=22-01-2013&newspaper=catedraTelefonicaUA@iuii.ua.es
p1
p2
```

Es justo lo que se espera.

Pues otro día después de pasar varias pruebas comprobé que no iba a la perfección así que desarrollé este nuevo código que era capaz ahora sí de detectar todos los casos de URL.

```cpp
void Tokenizador::TokenizarCasosEspeciales(const string& str, list<string>& tokens) const{
    string palabra;
    char caracter;
    bool esURL = false;
    int j;
    string especiales = "_:/.?&-=#@";
    bool delimitadoresURL[256] = {false};
    for(int i = 0; i < 256; i++){
        delimitadoresURL[i] = delimitadores[i];
    }
    for(char caracter : especiales){
        delimitadoresURL[static_cast<unsigned char> (caracter)] = false;    // solo está el resto de delimitadores exceptuando los validos en URL
    }
    for(int i = 0; i < str.length(); i++){
        caracter = str[i];
        if(delimitadores[static_cast<unsigned char> (caracter)] == true){   // es delimitador, fin de palabra
            if(palabra.empty() == false){
                tokens.push_back(palabra);
                palabra.clear();
            }
        }else{                              // no es un delimitador
            if(palabra.empty() == true){    // inicio de palabra
                if(str.substr(i, 4) == "ftp:" || str.substr(i, 5) == "http:" || str.substr(i, 6) == "https:"){
                    esURL = true;
                }
                if(esURL == true){  //  es una URL almacenamos todo el contenido
                    for(j = i; j < str.length(); j++){
                        caracter = str[j];
                        if(delimitadoresURL[static_cast<unsigned char>(caracter)] == false){    // no es delimitador (aunque sea uno especial // excepcion)
                            palabra += caracter;
                        }else{                              // fin de URL
                            if(palabra.empty() == false){
                                tokens.push_back(palabra);
                                palabra.clear();
                                break;
                            }
                        }
                    }
                    i = j;
                }else{      // no es URL
                    palabra += caracter;
                }
            }else{          // no es inicio de palabra
                palabra += caracter;
            }
        }
    }
    if(palabra.empty() == false){   // si la frase no termina con delimitador hay que añadirlo
        tokens.push_back(palabra);
        palabra.clear();
    }
}
```

Es básicamente más de lo mismo y aunque haya 2 bucles anidados aparentemente no es una complejidad cuadrática ya que se usa como auxiliar para avanzar el índice por lo que no volvería a recorrer esas posiciones de la string manteniendo así **complejidad O(n)**. La implementación es la misma que se hizo para tokenizar los ficheros normales sin casos especiales usando un array de booleanos pero ahora adaptado a los caracteres delimitadores que SÍ admiten las URLs.
Con el mismo main cambiando un token más sale esto:

```
antonio@ubuntu:~/Escritorio/Universidad/4/ei/EI/Entrega1$ ./memory ./main
Tokens obtenidos: 5
p0
http://intime.dlsi.ua.es:8080/dossierct/index.jsp?lang=es&status=probable&date=22-01-2013&newspaper=catedraTelefonicaUA@iuii.ua.es
p1
p2
ambiguo
Tiempo de ejecuci◆n: 1.4578e-05 s
```

## Detección del caso especial: Números decimales

Para ello según pone en la especificación al analizar las números decimales debemos cumplir con una serie de pasos:

1. Solo se considera número decimal si está compuesto únicamente por dígitos y . o ,
2. Si comienza por . o , se le añade un 0 delante
3. Si hay varios . o , seguidos solo se considera decimal el segundo →3..2 será 3 0.2

4. Si aparece % o $ al final del número y seguidos de espacio se almacenan como tokens separados

Comenzamos con la implementación en el método Tokenizar(const string&, list<string>&) porque este es el método usado para la tokenización en el ámbito de TokenizarListaFicheros. Actualización: lo intenté pero no tenía tiempo y no me iba así que se quedó aquí.

## Cálculo de la complejidad temporal del método Tokenizar en casos especiales

Para calcular la complejidad de este método hay que hacer referencia a qué hay dentro de él, como se puso la captura anteriormente cuando se terminó de implementar las URLs. Como no he podido completar todos los casos especiales haré el cálculo de la complejidad únicamente sobre este caso (por desgracia :c).
Habiendo un **bucle principal** para recorrer todo el string será **O(n)** y al usarlo de manera eficiente para recorrer las URLs pudiendo así **no** tener que **introducir otro bucle anidado** y **avanzar el índice una vez recorrida la URL** se queda con una **complejidad temporal O(n)**. Concluyendo como **caso peor O(n)** y como **caso mejor Ω(n)**. En ambos casos, haya casosEspeciales o no los haya obtendremos **O(n)**.

## Cálculo de la complejidad espacial del método Tokenizar en casos especiales

Para calcular la complejidad de este método hay que hacer referencia a qué hay dentro de él y cómo se inicializan las variables que se usan, como se puso la captura anteriormente cuando se terminó de implementar las URLs. Como no he podido completar todos los casos especiales haré el cálculo de la complejidad únicamente sobre este caso (por desgracia :c).
Haciendo uso de la herramienta ***memory*** podemos saber cual es el **pico más alto de memoria** como ya se hizo en capturas anteriores. Viendo el contenido de este método podemos observar que al no tener que hacer el recorrido del string de delimitadores tenemos una eficiencia espacial muy grande también. Situándonos en su **caso mejor** sería **complejidad espacial Ω(1)** ya que está hecho para que si tiene un input de todo delimitadores no guarde nada en memoria. La complejidad espacial en su **caso peor será O(n)** ya que tendrá que almacenar en tokens y en palabras tokens de longitud n. Como se puede ver en esta imagen tenemos un impacto en memoria muy pequeño.