**Procedimiento**

Implementación

Para la realización de un codificador/decodificador de texto en código ASCII a Base64, investigamos primero cuales eran las características y la forma en que opera cada uno de los códigos para poder determinar el mejor camino para codificar en ANSI C. A partir de la investigación realizada en la que observamos que podíamos optar por una filosofía al estilo UNIX en la que diseñaramos funciones con objetivos claros y simples, o bien podíamos organizarnos utilizando una filosofía menos ordenada en la que los distintos procesos estuvieran diseminados a lo largo de toda la función principal.  
 Si bien parece, por haberla calificado como clara y simple, que la filosofía UNIX es más sencilla de aplicar esto resulta a la inversa ya que requiere una gran planificación previo a la codificación y un gran nivel de detallismo en cuanto a las consideraciones, como ser excepciones, coherencia entre las distintas funciones, disposición y orden de jerarquía de las funciones. Mientras que en el caso de ir haciendo que los procesos se ejecuten a lo largo de toda la función principal tales inconvenientes de planificación detallada no se presentan.  
 Nosotros escogimos la filosofía al estilo unix ya que a pesar de que la planificación se tornara más engorrosa, los distintos procesos se realizarían en bloques bien definidos lo cual permitiría encontrar posibles errores o incluso mejorar posteriormente algún proceso en cuanto a rendimiento en rapidez o en almacenamiento.   
 Además, este modo de trabajo, posee el beneficio de producir pequeños módulos potencialmente útiles para otras aplicaciones futuras en las que se necesite realizar tareas ya codificadas en este programa. De esta forma, pensando a futuro, se ahorra tiempo de trabajo a largo plazo.   
 Sin embargo, esta filosofía no pudo ser adoptada para toda la modularización del programa, ya que ciertas tareas o bien requerían un acoplamiento por cohesión, o bien simplemente pensar el módulo de una tarea tan específica para aplicaciones futuras se tornaba demasiado complicado. Este es el caso, por ejemplo, del módulo que realiza un checkeo de la lectura en base 64. Si bien la lógica es utilizable para cualquier validación que se requiera hacer de un texto en base64, la codificación utilizada en esta implementación se comunica y está pensada a medida únicamente para la tarea específica que requería la aplicación es esa etapa.

Dispusimos una serie de procesos para el caso de codificación (pasar de ASCII a Base64):

1. Validar los comandos por línea de argumentos
2. Seleccionar un modo de operación (encoder/decoder)
3. Leer 3 caracteres ASCII y guardarlos en un buffer1.
4. Codificar los caracteres ASCII a 4 caracteres en Base64 y guardarlos en un buffer2.
5. Imprimir los caracteres, en Base64.

También organizamos una lista de procesos para el caso de decodificar (pasar de Base64 a ASCII):

1. Validar los comandos por línea de argumentos
2. Seleccionar un modo de operación (encoder/decoder)
3. Leer 4 caracteres Base64 y guardarlos en un buffer1
4. Validar la entrada de texto, es decir comprobar que los caracteres leídos estén acorde con las especificaciones requeridas por código Base64.
5. Decodificar los caracteres Base64 a caracteres ASCII y guardarlos en un buffer2.
6. Imprimir los caracteres, en ASCII.

Como se puede ver los primeros dos procesos son compartidos en ambos casos y luego de acuerdo a lo que el usuario ingrese, se decide, si es válido lo ingresado, si se ingresa al modo encoder o decoder.

Lectura  
 En cuanto al proceso de lectura de caracteres se optó entre las siguientes opciones:

* Realizar un arreglo tan grande como caracteres se hubieran ingresado, determinando el número de caracteres ingresados a partir del operador sizeof() o a partir de una interacion en la que un contador frene cuando se encuentre EOF.
* Realizar un arreglo de 3 elementos y guardar en el de a tres caracteres.

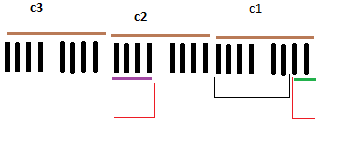
La primera opción fue descartada porque requiere de un gran espacio en memoria y además podría ocurrir que el texto ingresado ocupe más lugar en memoria de lo que esta puede almacenar, o también puede ocurrir que si el número de caracteres que se ingresó es mayor al máximo número que puede adoptar un unsigned long, entonces se produzca un error. Además no es necesario.  
 La segunda opción fue la elegida ya que al procesar de esta forma la entrada, se obtienen grupos de caracteres de 24 bits, número que es el mínimo común múltiplo de 6 y 8 y se los almacena en un arreglo de 3 elementos. Por otro lado al procesar los caracteres se predijo que pudiera existir la situación en la que no se ingresen 3 caracteres sino ninguno, 1 o 2. Teniendo, lo anteriormente expresado, en cuenta es que decidimos agregar a esta función la indicación del estado en que estaba operando la función, es decir el caso en que no se procesó nada (correspondiente al END), el caso en que se procesó un solo carácter (correspondiente al EXCEPTION\_1), el caso en que se procesaron dos caracteres (correspondiente al EXCEPTION\_2), y el caso en que se procesaron tres caracteres, caso óptimo (correspondiente a NORMAL\_WORKING).

Respecto a la implementación, hay que destacar el hecho de haber realizado una única función dedicada a la lectura tanto para el modo “encoder” como el modo “decoder”. Esta decisión fue tomada al realizar dos funciones distintas y notar que diferían únicamente en la cantidad de ciclos que realizaban y el estado (“status\_t”) que devolvían. Finalmente, se optó por un funión unificadora que reciba la dirección de un arreglo donde almacenar los caracteres y un entero con la cantidad de caracteres a leer, devolviendo por nombre la cantidad de caracteres efectivamente almacenados antes de encontrar la marca EOF. De este modo el estado lo decidiría otro proceso en base a la cantidad de caracteres pedidos y la cantidad de caracteres almacenados. Además, se obtiene una función sumamente útil para futuras aplicaciones a costas de un mínimo desperdicio de memoria (notar en el código fuente que la variable que recibe la cantidad de elementos a leer es del tipo size\_t, siendo que para este programa con un short alcanza).

Codificación

En cuanto al proceso de codificación se idearon dos alternativas las cuales en un principio se pensó que solo diferían en cuanto al orden de la lectura de los caracteres que se encuentran en el arreglo, pero luego se descubrió que no era tan asi. Las dos alternativas son:

* Procesa el primer elemento del arreglo, es decir el primer carácter ingresado, y lo ingresa en un buffer inicializado a cero y de tipo unsigned int y se lo desplaza a izquierda 16 lugares. Luego se ingresa el segundo carácter al buffer y se lo desplaza 8 lugares. Por último, se ingresa el tercer carácter al buffer y ya se tiene completo el llenado de este buffer. A continuación se define una máscara (0x3F) y se hace un AND de bits con el buffer y almacenando tal resultado en un buffer2, en el último elemento. Luego siguiendo el mismo principio, se repetirá lo anterior pero desplazando primero el buffer hacia la derecha de a 6 lugares y almacenando lo obtenido llenando el buffer2 desde el último elemento hasta el primero en ese orden. Para las excepciones la lógica es la misma salvo que en caso de estar en la excepción 1 el desplazamiento a izquierda será 4 lugares y se realizara por única vez siguiendo luego con el AND de bits y un desplazamiento a derecha (6lugares) y otro AND de bits, llenando de esta forma los elementos 1 y 0 del buffer2 respectivamente; los elementos 2 y 3 se llenaran con ´=´. En el caso de la excepción 2 la lógica vuelve a ser parecida a la de la excepción 1.
* En esta alternativa los caracteres son ingresados al buffer siguiendo la misma lógica explicada anteriormente pero ingresando primero el último elemento del arreglo que contiene los caracteres leídos y siguiendo así hasta el primero. Desde aquí ya ni se presentara como se tendría que codificar ya que con la siguiente explicación se entenderá que esta forma es sumamente complicada. Viendo el diagrama siguiente, podemos ver que lo primero que se tiene que procesar es lo negro, luego se tiene que procesar lo rojo, teniendo que hacer una máscara para el caso negro y otra para el subcaso verde del caso rojo, desplazar lo obtenido y sumarlo con una máscara (desplazada hacia la derecha) para el subcaso violeta del caso rojo. Como se puede ver esto es muy complicado y ni siquiera se terminó con la decodificación de dos caracteres Base64.



Impresión

# En lo que respecta a la impresión de los datos procesados, los módulos implementados son simples como se ve en los diagramas y en el código fuente, y mantienen el hilo filosófico que permitiría en otras aplicaciones imprimir tantos caracteres como número máximo tiene un size\_t.

**Conclusión**

A modo de conclusión, este grupo cree que el trabajo práctico ha sido un buen primer paso en diversas áreas de la programación. Estas áreas comprenden, el mundo de los protocolos, a través de la investigación que tuvo que realizarse en cuanto a la utilización de base64; la encriptación y compresión de datos en sistemas informáticos; la modularización de programas y sus criterios; la organización de código fuente en archivos separados por tareas(main, funciones, cabeceras); el trabajo en equipo y la coordinación mutua de un grupo de trabajo.