Práctica 4

Módulos en C

4.1 Tipos abstractos y modularización

La filosofía del diseño descendente de algoritmos puesta en práctica en la actividad anterior es un potente método de abstracción para la construcción de programas.

Al plantear la construcción de un programa de modo descendente, el programador introduce en cada nivel de refinamiento aquellos tipos de datos y las operaciones sobre los mismos que necesita para resolver el problema en los términos en los que éste se plantea en ese nivel, sin entrar a considerar la forma en la que se van a representar esos datos en la máquina, ni los algoritmos que usará en las operaciones que necesita para manipularlos.

De este modo, el programador evita tomar en consideración detalles que no son relevantes en el nivel de abstracción en el que trabaja en ese momento, reduciendo así la diferencia conceptual entre el razonamiento al que obliga la construcción del programa y las posibilidades expresivas del lenguaje de programación utilizado.

Cada nuevo tipo de datos que el usuario introduce, así como las operaciones con las que piensa manipular los valores del mismo, forman lo que llamaremos un *tipo abstracto de datos*[1].

Sucede que, cuando el problema a resolver tiene cierta complejidad, es fácil que su resolución exija procesar objetos de varias entidades abstractas. Y dado que cada una de estas entidades conlleva la introducción de un tipo abstracto de datos, puede resultar que en la solución final del programa abunden y se mezclen desordenadamente las definiciones –tanto de tipos de datos como de funciones y acciones– que pertenecen a diferentes tipos abstractos, que a su vez pueden haber aparecido en niveles de abstracción diferentes, haciendo todo ello muy difíciles la comprensión y el mantenimiento del programa.

Los *módulos* serán utilizados como *unidades de organización* en los problemas de diseño descendente: cada uno de los tipos abstractos de datos aparecidos en los sucesivos refinamientos será descrito dentro de su propio módulo, de forma autónoma e independiente del problema en el que apareció.

Esta descripción consistirá en una *colección separada de declaraciones y definiciones*. Esta separación responde a la necesidad de forzar, para mayor claridad y facilidad de uso,

[1] Se les califica de *abstractos* porque en el nivel de abstracción en el que son introducidos no es necesario conocer –insistimos en este importante punto– los detalles de su implementación, esto es, *no es necesario saber cómo son representados sus valores internamente en la máquina ni qué algoritmos se ha decidido emplear en la manipulación.*

la separación entre los detalles de implementación y los detalles de uso de los elementos que provee el módulo.

Así, dentro de un módulo habrá siempre una parte formada por declaraciones que informe del tipo de dato y de las operaciones asociadas al mismo que ese módulo proporciona a quien lo vaya a incluir en sus programas, y una parte formada por definiciones que describan cómo son representados y manipulados internamente en la máquina los valores pertenecientes a ese tipo de dato.

La parte formada por declaraciones recibe el nombre de *lista de exportaciones* del módulo y la parte formada por las definiciones se llama *implementación* del módulo.

De forma acorde con la denominación dada a la parte de declaraciones de un módulo, se dirá que éste *exporta* los elementos cuyas declaraciones aparecen en la lista de exportaciones. Asimismo, se dice que un algoritmo *importa* los elementos puestos a disposición en la lista de exportaciones de un módulo, cuando lo incorpora añadiendo su nombre dentro de una sección **usa** ... **fusa** que debe aparecer tras la cabecera.

Pero también puede ocurrir que sea un módulo el que necesite importar los elementos exportados por otro u otros módulos. Para ello se incluye el nombre de los módulos cuyos elementos se quieren importar en la llamada *lista de importaciones* del módulo.

Para controlar todo lo anterior, se emplea una sintaxis especial que permite distinguir las diferentes secciones que forman un módulo. Estas aparecen en el siguiente esquema:

```
modulo ⟨nombre⟩;
    importa
        # Lista de importaciones:#
        # relación de nombres de módulos de los que se importa algo.#
    fimporta
    exporta
        # Lista de exportaciones.  #
        # Es una relación de elementos exportables:#
        # constantes, tipos, variables, acciones y funciones.#
    fexporta
    implementación
        # Definiciones de constantes, tipos, acciones y funciones.#
    fimplementación
fmódulo
```

4.1.1 Un ejemplo conocido: el tipo abstracto *palabra*

Como primer ejemplo, vamos a construir un módulo llamado *palabras*, que exporte el tipo abstracto de datos *palabra* visto en el tema anterior. En este módulo se recogerán todas las operaciones definidas sobre el tipo *palabra* vistas en los ejemplos y problemas de ese tema.

```
modulo palabras;
    exporta

        Entrada:
        Requisitos:
        Salida:  p, una palabra inicializada a 0 letras;
        accion prepararPalabra(sal p:palabra);
```

```
    Entrada:  p, una palabra y c un caracter
    Requisitos:
    Modifica:  p, incorpora c como ultimo caracter de p;
    accion agregarCaracter(e/s p:palabra; ent c:  caracter);

    Entrada:  p, una palabra, el entero i y c un caracter
    Requisitos:  i es menor o igual que la longitud de p
    Modifica:  Cambia por c el caracter i-esimo de p;
    accion modificarCaracter(e/s p:palabra;ent i:entero;ent c:caracter);

    Entrada:  p, una palabra y el entero i
    Requisitos:  i es menor o igual que la longitud de p
    Salida:  Devuelve el caracter i-esimo de p;
    funcion consultarCaracter(p:palabra; i:  entero) dev c:  caracter;

    Entrada:  p, una palabra;
    Requisitos:
    Salida:  long, la longitud de p
    funcion longitudPalabra(p:palabra) dev long:  entero ;

    Entrada:  p, una palabra;
    Requisitos:
    Salida:  el booleano b es verdadero si y solo si p no tiene letras
    funcion esPalabraVacia(p:palabra) dev b:  booleano ;

    Entrada:  p1 y p2, dos palabras;
    Requisitos:
    Salida:  el booleano b es verdadero si y solo si p1 y p2 son iguales
    funcion sonPalabrasIguales(p1, p2:palabra) dev b:  booleano;

    Entrada f:  fichero de caracter;
    Requisitos:  El fichero no esta vacio y pd(f) = αM y α ≠ ε;
    Modifica:  f avanzando el elemento distinguido hasta el principio
            de la siguiente palabra;
    Salida:  p (entero) contiene la primera palabra de α;
    accion leerPalabra(e/s f:  fichero de caracter; sal p:palabra);

fexporta

implementacion
    const
        MAX = 100;
    fconst

    tipo
        palabra = tupla
            letras:  tabla[1..MAX] de caracter;
            long:0..MAX;
        ftupla;
    ftipo

    accion prepararPalabra( sal p:palabra);
        p.long:= 0
    faccion;

    accion agregarCaracter(e/s p:palabra; ent c:  caracter);
```

```
    si longitudPalabra(p) < MAX ⟶
        p.long:= p.long +1;
        p.letras[p.long]:= c
    [] longitudPalabra(p) ≥ MAX ⟶
        continuar
    fsi
faccion;

accion modificarCaracter(e/s p:palabra; ent i:  entero; ent c:  caracter);
    p.letras[i]:= c
faccion;

funcion consultarCaracter(p:palabra; i:entero) dev c:  caracter;
    c:= p.letras[i];
    dev c
ffuncion;

funcion longitudPalabra(p:palabra) dev long:  entero;
    long:= p.long;
    dev long
ffuncion;

funcion esPalabraVacia(p:palabra) dev b:  booleano;
    b:= (longitudPalabra(p) = 0);
    dev b
ffuncion;

funcion sonPalabrasIguales(p1, p2:palabra) dev b:  booleano;
    var
        i:1..MAX;
    fvar
    si longitudPalabra(p1) = longitudPalabra(p2) ⟶
        i:= 1;
        mientras (i<longitudPalabra(p1)) y (consultarCaracter(p1, i)
                    = consultarCaracter(p2, i)) hacer
            i:= i + 1
        fmientras;
        b:= (consultarCaracter(p1, i) = consultarCaracter(p2, i))
    [] longitudPalabra(p1) ≠ longitudPalabra(p2) ⟶
        b:= falso
    fsi;
    dev b
ffuncion;

accion saltarBlancos(e/s f:  fichero de caracter; sal c:  caracter);
    leer(f, c);
    mientras no fdf(f) y (c = ' ') hacer
        leer(f, c)
    fmientras
faccion;

accion copiarLetras(e/s f:  fichero de caracter; e/s c:  caracter;
            sal p:palabra);
    mientras no fdf(f) y (c ≠ ' ') hacer
        agregarCaracter(p, c);
        leer(f, c)
```

```
            fmientras;
            si c ≠ ' ' ⟶
                agregarCaracter(p, c)
            [] c = ' ' ⟶
                continuar
            fsi
        faccion;

        accion leerPalabra(e/s f:  fichero de caracter; sal p:palabra);
        var
            c:  caracter;
        fvar
            prepararPalabra(p);
            si no fdf(f) ⟶
                saltarBlancos(f, c);
                copiarLetras(f, c, p)
            [] fdf(f) ⟶
                continuar
            fsi
        faccion;
    fimplementacion
fmodulo
```

Puede verse que no todas las acciones y funciones del módulo se exportan. En concreto, las acciones `saltarBlancos` y `copiarLetras` se utilizan como acciones auxiliares en la implementación de acciones de nivel superior por lo que no son exportadas. Se suele decir que su definición es *local* ya que queda confinada en la parte de implementación del módulo.

El algoritmo que obtiene la palabra más larga de un fichero de caracteres queda reducido ahora a lo siguiente:

```
Entrada:  f, un fichero de caracteres;
Requisitos:  f no esta vacio;
Salida:  pMax es la primera aparicion en f de una palabra de longitud maxima
algoritmo palabraMasLarga;
    usa
        palabras;
    fusa
    var
        p:  palabra;
    fvar
    abrir_lec(f);
    prepararPalabra(pMax);
    mientras no fdf(f) hacer
        leerPalabra(f, p);
        si longitudPalabra(p) > longitudPalabra(pMax) ⟶
            p_max:= p
        [] longitudPalabra(p) <= longitudPalabra(pMax) ⟶
            continuar
        fsi
    fmientras;
    cerrar(f)
falgoritmo
```

4.2 Traducción a lenguaje C

4.2.1 Definición de módulos en lenguaje C

En C no existe el concepto módulo, pero podemos crearlo utilizando el concepto de biblioteca (librería). C nos permite compilar ficheros que no contengan un programa completo, es decir, que no contengan la función `main`. Posteriormente podemos usar lo que se haya definido o declarado en dicho fichero dentro de otro programa. Para ello debemos conseguir que el fichero que contiene el programa conozca los nombres definidos o declarados en el primer fichero e informarle al `linker` donde se encuentra el fichero que contiene esas definiciones compiladas. Como ve el proceso implica varios pasos, así que vamos a analizarlos por partes.

Creación de un modulo en C

Para programar el módulo `nombre` en C debemos crear dos ficheros `nombre.h` y `nombre.c`. El primero de ellos contiene la parte de exportaciones del módulo, es decir, las definiciones de los tipos que se desean exportar junto con los prototipos de las funciones que se desean exportar[2]. Recuerde que un prototipo es la cabecera de la función eliminando los nombres de los parámetros formales.

En algunas ocasiones nos vemos obligados a incluir en el `.h` cosas que no queríamos exportar. Por ejemplo, si queremos exportar un tipo estamos obligados a incluir en el `.h` su definición. A veces también tenemos que exportar parte de las importaciones del módulo. Como es sabido, en C no podemos utilizar nada antes de que se conozca su declaración. Si estamos exportando un tipo para cuya definición es preciso utilizar algo importado es, por tanto, necesario poner la importación antes de la definición del tipo. Lo más adecuado para proteger el código es incluir en el `.h` sólo aquellas cosas a las que nos obligue el compilador.

El segundo fichero, el `.c` va a contener la parte de importaciones y de implementación. Debe comenzar con una directiva del macroprocesador que incluya el `.h` que contiene las exportaciones. Para ello pondremos `#include ‘‘nombre.h’’` [3]. A continuación suelen ir las importaciones. Para importar un módulo debe incluirse su *header*, es decir, el `.h` de los módulos que queramos importar. Esta inclusión se realiza nuevamente utilizando `#include ‘‘otroNombre.h’’` si se está incluyendo un *header* no estándar de C o `#include <otroNombre.h’>` si es un *header* estándar. Por último están las definiciones de las funciones que se exportan junto con todo el código adicional que sea necesario.

Una vez tenemos estos ficheros, debemos compilarlos. Para ello debemos ejecutar el compilador con la opción `-c`, es decir ejecutar `gcc -c nombre.c`. La opción `-c` le indica al compilador que el fichero no es un programa completo. Ante ella el compilador se limita a comprobar que el código es correcto, a comprobar que no se utiliza ningún nombre que no se haya declarado y a generar un fichero objeto, que se llamará `nombre.o` que contiene el código de las funciones que se han implementado en el `.c`

[2]C También permite exportar otras cosas como macros, constantes y variables. No hay problema con las macros, aunque hay que tener cuidado con los nombres que reciben para que no haya conflictos si importamos muchos módulos. Está fuertemente desaconsejado exportar constantes o variables.

[3]Esto si el `.h` está en el directorio donde estamos compilando. En caso contrario debemos incluir la ruta del `.h`, fuertemente desaconsejado, o indicar dicha ruta en la línea de comando cuando compilamos mediante la opción `-I`*`ruta`*

Uso de un módulo en C

Para usar un módulo en un programa C basta con incluir su fichero de cabecera mediante la directiva `#include ‘‘nombre.h’’` y suministrarle al compilador el fichero objeto `nombre.o`. Esto último se puede hacer de diversas maneras:

- Si es un módulo de la librería estándard de C puede no ser necesario hacer nada. Muchos de los módulos de esta librería son accesibles por defecto. Es decir, basta con hacer `gcc programa.c`
- Para algunos módulos de la librería estándard de C se utiliza la opción `-lx` donde x es una cadena que identifica que fichero de los que contiene dicha librería se desea suministrar. Por ejemplo con `gcc programa.c -lm` estaríamos suministrando las funciones matemáticas de la librería estándar, que se encuentran en el fichero libm.a junto con el resto de librerías estándard de C.
- Podemos indicar las rutas de los fichero objeto que se desean en la línea de compilación. Es decir con el comando `gcc programa.x nombre.o`, si el fichero `nombre.o` está en el directorio actual, estaríamos suministrándole al compilador el código de todo lo exportado por el módulo `nombre`
- Podemos crear un directorio donde depositar nuestros ficheros `.o` e indicar en la línea de compilación, con la opción `-L`*`ruta`* cuál es dicho directorio. Esta opción es ms adecuada si previamente hemos juntado nuestors `.o` en una librería real[4].

Hay que destacar que no es exactamente el compilador el que usa dichos ficheros. La compilación tiene diversas fases. La última es la de enlazado. El enlazador (o *linker*) conecta las llamadas a nombres que no están declarados en el programa principal con la definición de ese nombre en los ficheros `.o` o librerías que se le suministre.

Hay que tener en cuenta que el linker trata de modo diferente a las librerías que a los archivos `.o` que le suministre por línea de comando. Para empezar, los `.o` no pueden tener ningún nombre de función, tipo o variable global repetido y las librerías sí. Además, el linker busca cada nombre siguiendo el orden en el que se le han suministrado los `.o` y bibliotecas. Por ejemplo, si compilamos `gcc programa.c modulo1.o -lm` y `programa.x` usa la función `f1`, el linker busca el código de `f1` primero en `modulo1.o` y sólo si no lo encuentra lo busca en `libm.a`. SIn embargo, si hacemos `gcc programa.c -lm modulo1.o` lo busca en `libm.a` y , si lo encentra, ni siquiera mira `modulo1.o`. El orden de la línea de comandos es fundamental, pero también lo es que se tenga mucho cuidado con los nombres para evitar conflictos.

El problema de las importaciones cíclicas

Como ya le he contado, la directiva `include` lo que hace es copiar el archivo incluido. Si, por algún, motivo este archivo incluye a otro que también le incluye a él, llegaríamos a un bucle infinito. Otro problema que podemos tener es que incluyamos dos ficheros que ambos incluyen a un tercero. En tal caso, los nombres de ese fichero estarían declarados

[4]Para tener más claro el concepto de qué es y cómo se usa una librería le recomiendo que lea la página http://www.chuidiang.org/clinux/herramientas/librerias.php , aunque no va a ser necesario para lo que va a hacer en este curso

multiples veces, con lo que el programa no puede compilar. Nada de esto ocurre si las importaciones las escribimos en los `.c`. A pesar de ello, es una costumbre lamentable, pero muy extendida, incluirlas en los `.h`. Para que ningún usuario de sus `.h` sufra problemas por ello, se emplean otras directivas del macroprocedador de C.

Podemos asociar a cada *header* una macro y controlar su inclusión mediante dicha macro. Para ello basta con escribir los *headers* con la siguiente estructura

```
#ifndef NOMBRE_MACRO
#define NOMBRE_MACRO
exportaciones
#endif
```

La primera vez que el macroprocesador lee el header `NOMBRE_MACRO` no está definida, por lo que se define y se tienen en cuenta las exportaciones. Si, por cualquier causa, el macroprocesador vuelve a leer el mismo header, `NOMBRE_MACRO` ya está definida, por lo que se obvia todo hasta el `#endif`.

Una ayuda: la utilidad `make`

Con todo lo dicho, queda claro que construir un programa modular implica ejecutar muchos comandos, utilizando, además, diferentes opciones. La utilidad `make` se crea para organizar todo este proceso. Esta utilidad, que viene por defecto en los sistemas Unix/Linux. permite describir mediante un fichero como se construye la aplicación. Cuando ejecutamos `make`, la utilidad busca en el directorio acual un fichero llamado `makefile` y, si no lo encuentra, otro llamado `Makefile` y ejecuta las ordenes que diche fichero le indique

Los ficheros que utiliza `make` son ficheros de texto en los que hay una secuencia de reglas. Una regla tiene tres partes:

```
objetivo:  dependencias
    comandos
```

donde hay que destacar que las `dependencias` deben estar en la misma línea que el objetivo y que los comandos son una secuencia de líneas que tienen que empezar con un tabulador (nunca una secuencia de blancos, tenga cuidado con lo que hace su editor). El `objetivo` puede ser un fichero que queremos crear o una etiqueta que no se refiere a ningún fichero. Las `dependencias` indican que otros ficheros u objetivos previos son necesarios para lograr el `objetivo`, mientras que los comandos son la secuencia de comandos que se deben ejecutar para conseguir el objetivo a partir de sus dependencias.

Al ejecutar make le indicamos cual es nuestro objetivo. El comando comprueba si es necesario volver a crear nuestro objetivo comprobando, en el caso de que sea un fichero, si se han actualizado los ficheros de los que depende después de la última vez que se construyo el objetivo y, si es así ejecutará los comandos que hayamos indicado para construir el objetivo.

Es decir, podríamos tener, por ejemplo una regla que fuese:

```
programa:  archivoFuente.c modulo1.c
    gcc -c modulo1.c
    gcc archivoFuente.c modulo1.o -o programa
```

Si ejecutamos `make programa` despues de cambiar algo de `modulo1.c` make detectaría que debe reconstruir `programa` y ejecutaría los dos comandos que permiten construirlo

Cabe destacar que podemos incluir comentarios. Un comentario en un `makefile` no es más que una línea que comienza con #). También podemos definir variables (que funcionan de modo similar a las macros de C).

El comando make es potente y complejo. Os dejo el manual completo en el directorio de material para la práctica 4. Obviamente no es necesario que lo leáis ahora. Para empezar a utilizar make basta con lo que os he contado. Si queréis un poco más podéis mirar la entrada de make en la wikipedia en castellano. Y si aún queréis profundizar un poco más pero sin cansaros, podéis leer la misma entrada en la versión en inglés.

Un ejemplo

En /Recursos/Laboratorio/Material para la práctica 4 os he dejado una serie de ficheros que podéis usar como ejemplo de como se utiliza todo lo dicho. He creado un módulo[5] que crea un tipo fichero con las mismas características que los ficheros de lenguaje algorítmico a partir de los ficheros de C. El modulo está repartido en dos archivos. Las exportaciones en `ficheroDeCaracteres.h` y la implementación en `ficheroDeCaracteres.c`. He creado como ejemplo de uso de módulos el fichero `copiaFichero.c` que importa el módulo `ficheroDeCaracteres` y lo utiliza para hacer un algoritmo muy sencillo de recorrido que copia el fichero de caracteres que se le indique en otro fichero llamado copia. También os he dejado un ejemplo muy corto de `makefile` para que comprobéis el uso de `make`. Si ejecutais `make` se intenta completar el primer objetivo (la ejecución del programa `copiaFichero`) dando todos los pasos que sean necesarios para ello. Si el programa no ha sido compilado o se detecta que se ha modificado algo, vuelve a compilar lo necesario para construir una nueva versión. Si ejecutais alguno de los otros dos objetivos (haciendo `make copiaFichero` o `make ficheroDeCaracteres.o` construireis el ejecutable (sólo si es necesario) o el `.o` (sólo si es necesario).

Uso de librerías

Cabe destacar que en los programas escritos hasta ahora ya se ha hecho uso de algunos modulos que el propio compilador `gcc` nos ofrece. Por ejemplo, todas las funciones de entrada/salida. En este sentido hay que destacar que nos podemos encontrar con módulos que se pueden usar con solo importar sus ficheros de cabecera (como ocurre con las funcones de entrada/salida) y con otros para los que, además, es necesario indicarle al linker donde están los `.o` (`.a`, `.so` o `.lib` si están dentro de una librería), como es el caso de la librería matemática de C.

4.3 Actividades a realizar

1. Reescríbanse el programa que en el tema anterior calculaba el número de apariciones de la primera palabra (`cuenta.c`), esta vez usando el módulo palabras (`palabras.c`).

[5] que funciona y facilitaría la programación de los programas con ficheros en C pero que os recomiendo que no uséis si queréis pasar por programadores de verdad

2. Escríbase un programa (`crono.c`) que implemente un cronómetro utilizando la librería `ncurses` de C. Esta librería nos permite gestionar la ventana de comandos de modo que se pueden crear interfaces de usuario con características un poco más potentes. La interface del programa debe cumplir los siguientes requisitos:

   (a) Se debe abrir una ventana de 20 columnas por 10 filas con fondo en rojo y borde negro. En la primera fila, centrado, debe aparecer el nombre del autor. En la fila 5 y centrada, debe aparecer la cuenta con el formato MM:SS::D (minutos, segundos, décimas de segundo). En las filas 7, 8 y 9 deben aparecer los mensajes 's: star/stop', 'r: reset' y 'q: quit' respectivamente. Todos los textos deben aparecer en blanco.

   (b) Al pulsar la tecla 's' debe ponerse en marcha la cuenta y al pulsarla de nuevo esta debe parar. Cuando la cuenta esté parada, es posible ponerla a cero pulsando la techa 'r' y también es posible salir del programa pulsando 'q'.

   (c) Cuando la cuenta está en marcha debe representarse siempre en la misma posición de la ventana (fila 5 y centrada) y formato (MM:SS::D) pero su texto debe aparecer en verde. Cuando la cuenta está parada debe aparecer en blanco.

   (d) Al salir del programa deben restablecerse las ventanas originales y los colores normales de fondos y de textos.

   Para hacer el programa se debe crear un módulo llamado `acciones` como se indica a continuación

4.3.1 Esquema para el cronómetro

Conviene hacer dos cosas. Un módulo `acciones.c` y un programa `crono.c`. Aquí puede encontrar parte del módulo (sólo tiene algunas implementaciones, el resto debe pensarlas) y el algoritmo del programa.

```
modulo acciones;
    importa
        ncurses; // accede al modo semigráfico de pantalla y WINDOWS
        unistd; // para acceder a funciones de control de tiempo
        stdlib; // para acceder a exit y a funciones de control de tiempo
        stdbool; // para accder a las constantes true y false
    fimporta
    exporta
        const
            WDIMX= 30; WDIMY=10; //Dimensiones de la ventana
        fconst
        accion titulo(sal w:  WINDOW);
        accion normal( e/s w:  WINDOW);
        accion parado( e/s w:  WINDOW; ent min, seg, dseg:  entero);
        accion acero( e/s w:  WINDOW; sal min, seg, dseg:  entero);
        accion contar(e/s w:  WINDOW; e/s min, seg, dseg:  entero);
    fexporta
    implementacion
        accion titulo(sal WINDOW);
            //iniciar ncurses
            //capturar dimensiones de la ventana actual
            //hacer que la entrada vaya a programa, no a buffer
```

```
            //hacer que la entrada no vaya a pantalla
            //crear la ventana
            //ponerle el borde a la ventana
            //Activar modo de color
            //Definir los pares de colores (letra, fondo) a usar
            //Hacer que el cursor sea invisible
            //Elegir patron cores de parado
            //Crear la pantalla inicial con autor y guía de utilización
            //Sacar la pantalla creada
        faccion;
        accion normal( e/s w:  WINDOW);
            //Restablecer colores normales
            //Eliminar la ventana
            //salir del modo ncurses
        faccion;
        accion parado( e/s w:  WINDOW; ent min, seg, dseg:  entero);
            //Activar los colores de textos en parado
            //Mostrar valor de min, seg, dseg en su posicion y formato
        faccion;
        accion acero( e/s w:  WINDOW; sal min, seg, dseg:  entero);
            //poner a cero contadores
            //parado(w, min, seg, dseg);
        faccion accion contar(e/s w:  WINDOW; e/s min, seg, dseg:  entero);
            var
                c:  caracter;
            fvar
            //Activar colores de textos en marcha
            repetir
                //suspender la espera por instrucciones de entrada
                repetir
                    //Esperar 1 decima de segundo
                    dseg:= dseg +1;
                    seg:= seg + (dseg div 10);
                    min:= min + (seg div 60);
                    dseg:= dseg mod 10;
                    seg:= seg mod 60;
                    //Mostrar valor de min, seg, dseg en su posición y formato
                hasta que se lea una tecla en c frepetir;
                //activar la espera por instrucciones de entrada
            hasta c='s' frepetir;
            parado(w, min, seg, dseg)
        faccion;
    fimplementacion
fmodulo

algoritmo crono;
    var;
        min, seg, dseg:  entero;
        c:  caracter;
        w:  puntero a WINDOW //el algoritmico no permite declarar esto bien
    var;
    titulo(w);
    acero(w, min, seg, dseg);
    repetir
        //leer una tecla en c
        si c = 's' ⟶
```

```
            contar(w, min, seg, dseg)
        [] c = 'r' ⟶
            acero(w, min, seg, dseg)
        [] (c ≠ 's') o (c ≠ 'r') ⟶
            continuar
        fsi
    hasta c='q' frepetir;
    normal(w)
falgoritmo
```

4.3.2 Sobre ncurses

La librería `ncurses` proporciona funcionalidad que permite escribir interfaces de usuario basadas en texto con un gran control sobre el aspecto de la pantalla. Es una evolución de la mucho más antigua `curses`. Forma parte de la especificación del estándard Unix (o sea, cualquier sistema Unix debería tenerla como una de las librerías del sistema). No ocurre lo mismo en Windows[6]. Por tanto, un programa que utilice `ncurses` no puede correr en Windows.

La librería define gran número de cosas, como puede ver en la página de The Open Group https://pubs.opengroup.org/onlinepubs/7908799/cursesix.html. De hecho, puede encontrar diversos libros que explican como usar de un modo potente `ncurses`. Como leer toda la documentación disponible es casi imposible, puede utilizar como introducción el post https://wozgeass.wordpress.com/2009/09/13/c-con-ncurse/ y su segunda parte https://wozgeass.wordpress.com/2009/11/16/c-con-ncursessegunda-parte/ . También me parece muy interesante, aunque densa, la información que se muestra en la siguiente página http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/

En cualquier caso, la documentación de las funciones de `ncurses` están en el `man` de linux. Para hacer esta práctica necesita usar, algunas funciones de `ncurses` y entender que significa el tipo `WINDOW *`. A continuación le explico algo sobre ellas, pero para la documentacin completa debe consultar las fuentes indicadas:

Cuando iniciamos `ncurses` se crea una variable denominada `stdscr` de tipo `WINDOW *`. Las variables de este tipo son representaciones lógicas de la ventana de texto en la que se está trabajando. La variable `stdscr` representa la centana de comandos en la que se está trabajando y tiene sus dimensiones (número de filas y de columnas). Las funciones de `ncurses` escriben sobre variables de tipo `WINDOW *` y permiten sacar por pantalla el contenido de una de dichas variables. La funcionalidad exportada por `ncurses` incluye:

- `int attrset(int attrs);` aplica los atributos `attrs` a la ventana actual. Los atributos pueden ser, entre otros mapas de colores definidos mediante `init_pair();`. Para ello debemos usar como atributo `COLOR_PAIR(n)` siendo `n` el par de los definidos que queramos elegir. También hay atributos predefinidos como `A_BLINK`, `A_BOLD`, `A_DIM`, `A_NORMAL`, `A_REVERSE`, `A_STANDOUT` y `A_UNDERLINE`.

- `int box(WINDOW *win, chtype verch, chtype horch);` rodea `win` con un borde. Las otras dos entradas son códigos usados para representar el tipo de borde. Pruebe con 0, 0.

[6] Aunque para Windows existe `conio` que es similar

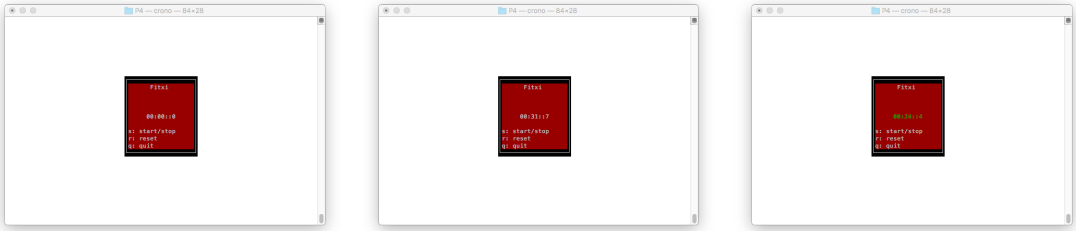
- `int cbreak(void);` hace que lo escribamos vaya directo al programa, no a un buffer intermedio.
- `int curs_set(int visibility);` define la visibilidad del cursor. Puede ser invisible (0), el normal (1) o más visible (2).
- `int delwin(WINDOW *win);` elimina la ventan lógica `win`.
- `int endwin(void);` cierra `ncurses` y recupera la ventana en la que se estaba trabajando al llamar a ncurses.
- `void getmaxyx(WINDOW *win, int y, int x);` carga en y y x el tamao de la ventana win.
- `void getyx(WINDOW *win, int y, int x);` recoge en y y x la posición actual del cursor.
- `bool has_colors(void);` deveulve verdadero si el terminal soporta colores.
- `int init_pair(short pair, short f, short b);` permite definir parejas de colores letra/fondo. El valor `pair` es el que identificara la combinación de colores. `f` y `b` tienen que ser colores definidos por `ncurses`. Los colores definidos son `COLOR_BLACK`, `COLOR_BLUE`, `COLOR_GREEN`, `COLOR_CYAN`, `COLOR_RED`, `COLOR_MAGENTA`, `COLOR_YELLOW` y `COLOR_WHITE` aunque el resultado no siempre coincide con el color esperado.
- `WINDOW * initscr(void);` Inicializa ncurses y devuelve `stdscr` como una copia lógica del terminal que se esté usando, es decir, el mismo tipo de terminal y el mismo tamaño. Para abrir otro tipo de terminales debe usarse `SCREEN *newterm(char *type, FILE *outfile, FILE *infile);`, pero es mucho más complejo.
- `int mvwprintw(WINDOW *win, int y, int x, char *fmt, ...);` mueve el cursor de la ventana indicada a la posición indicada y escribe allí lo indicado por `fmt`.
- `WINDOW *newwin(int nlines, int ncols, int begin_y, int begin_x);` crea una nueva ventana de `nlines` filas y `ncols` columnas posicionada en la fila y columna indicadas por `begin_y` y `begin_x`.
- `int nodelay(WINDOW *win, bool bf);` indica si las funciones de lectura de teclado deben bloquear la ejecución de la ventana `win` o no. Obviamente para que no haya espera `bf` debe ser verdadero.
- `int noecho(void);` deshabilita el echo, es decir, lo que escribimos no se ve en la ventana.
- `int printw(char *fmt, ...);` escribe en la ventana actual en la posición actual del cursor lo apuntado por `fmt`. El apuntador `fmt` funciona como el de printf();
- `int refresh(void);` refresca la ventana actual, es decir, escribe en la ventana real lo que hayamos escrito en la lógica que la representa
- `int start_color(void);` inicializa el uso de colores en la terminal actual.

- `int wattron(WINDOW *win, int attrs);` aplica los atributos `attrs` a la ventana `win`. Los atributos pueden son los mismos que se indicaron en `attrset();`.
- `int wgetch(WINDOW *win);` lee un caracter del buffer conectado con `stdin`. Si no hay nada y estamos en modo `nodelay` devuelve `ERR`.
- `int wmove(WINDOW *win, int y, int x);` miueve el cursor de la ventana indicada a la posición indicada. El movimiento no se ve hasta que no se haga un refresco.
- `int wrefresh(WINDOW *win);` refresca la ventana `win`, es decir, escribe en la ventana real lo que hayamos escrito en la ventana lógica que la representa.

Puede observar que también le digo que importe unistd y stdlib. `stdlib.h` nos permite acceder a las funciones de la librería estandard de C. Esta librería es una de las estandard y puede localizar su documentación facilmente en la web. La necesitará para la funcioón `exit()`. `unistd.h` le permite utilizar la función `usleep()` con la que podrá contar el paso del tiempo. La documentación de esta función tampoco es difícil de encontrar. Ninguna de ellas necesita que haga nada especial a la hora de compilar[7].

4.3.3 Resultados esperados

Le pego aquí un par de imágenes que muestran el resultado esperado (en un terminal que inicialmente tiene el fondo blanco), tanto inicalmente como en parado como en marcha:



Amplio un poco en la siguiente página la figura de en marcha

[7]No ocurre lo mismo con ncurses, aunque le dejo que localice por su cuenta como debe compilar un programa que utilice esta librería


P4 — crono — 84×28
Fitxi
00:24::4
s: start/stop
r: reset
q: quit