Un primer paso hacia *curses*

Interfaces Simples

Tal vez uno de los principales motivos para utilizar un interfaz de texto sea uno que dicta el sentido común de la economía. Más o menos nuestra máxima podría rezar así: "¿Para qué más?". Efectivamente: ¿para qué? ¿Qué necesidad tiene un almacén de piezas de recambio de automóviles de un Pentium VII a un millón de gigaherzios con pantallas de plasma de tropocientos pixels y mogollomiles de colores? Tal vez sería más realista escoger máquinas más básicas, incluso de las consideradas obsoletas, e invertir lo que se ahorre en hardware (y software, como después veremos) en otro sitio. POR PAUL C. BROWN

Mucho más práctico, digo, es tener una pantalla en blanco y negro, con buen contraste, un ordenador sencillito. Sinceramente, los interfaces amigables con muchos colorines e iconos de diseño están sobrevalorados. No sólo eso, sino que pueden inducir a confusión. Si se piensa racionalmente, muchas aplicaciones (la mayoría) no necesitan de interfaz gráfico. Perfectamente podrían apañarse con un interfaz de botones, menús y ventanas basadas en texto. Amén de resultar menos exigente con el hardware, también serían más claros y más sencillos de utilizar. Precisamente para desarrollar interfaces para aplicaciones de estas características existe *curses*, una librería que facilita enormemente la creación de ventanas, menús y widgets en terminales de texto. Por supuesto que las librerías *curses* están disponibles para la mayoría de los Unixes, incluyendo Linux y suelen incorporarse con casi todas las distribuciones e incluso existen versiones para otras plataformas lo que asegura, hasta cierto punto, la portabilidad del código (véase [1]). En este primer capítulo destinado a *curses* vamos a ver como emplear la librería imbuida en la infraestructura de una aplicación desarrollada en C++. Una advertencia: el C++ no es el entorno natural de *curses* y, al menos en un caso, no conseguiremos unas compilaciones cien por cien limpias (es decir, sin advertencias), si bien esto es más bien un problema del compilador *g++* que del código de muestra o de un defecto de *curses*, como después veremos.

Aplicándose a la tarea

Tal vez la manera más racional de enfocar el desarrollo de la aplicación (al menos es la que funciona para mí) es el de concebirlo desde arriba hacia abajo. Es decir, piensa en como quieres que se vea y ya te encargarás del trabajo sucio después. Normalmente esto implica empezar con la función *main()* y reducirlo a unas pocas llamadas – y cuando digo unas pocas, quiero decir unas poquísimas. Si la función *main()* puede contener como mucho 10 líneas de código, es que vamos bien. Después vamos implementando clases de las más generales a las más específicas, siendo estas últimas las que de verdad interactúan con las librerías específicas, curses en este caso. Así que, yo ¿cómo quería que se viese? Pues quería que en *main()* se inicializase un objeto x de una clase, llamémosle *application*, y que esa clase se ocupase de mostrar la ventana principal de la aplicación, colocara los menús etc. El resto de *main()* se ocuparía con un bucle *while* que se encargase de procesar los tecleos y pasarlos a la clase *application* y, al final, cuando se pulsase la tecla que precipitase la salida del bucle, se llamase al destructor del objeto y todo se recogiese limpia y ordenadamente. Sencillo. El corazón de dicha clase se puede apreciar en el Listado 1. No es muy impresionante ¿verdad? Eso es por que la mayor parte del "trabajo sucio" se lleva a cabo en w*indow*, que al ser heredado por *application*, recibe una llamada a su constructor cuando se crea un objeto *application*. Y es en el constructor de *window* donde se inicializa toda la infraestructura de curses, preparándola para mostrar las ventana. Se puede apreciar la herencia de *window* por parte de *application* en el fichero de inclusión application.h, no listado en este artículo, pero que puede ser descargado desde la web de la revista en [2]. Un poco más adelante volveremos sobre la clase *window*. Volviendo al constructor de *application*, lo primero que se

hace es realizar una llamada al macro de fabricación casera *makeString()* que sirve para convertir una cadena con un número de argumentos opcionales en una sola cadena tipo *string* que se utilizará como título (asignado, pues, a la propiedad de la clase *title*). Veremos más sobre este macro en la sección Parámetros Indefinidos más abajo. Lo siguiente es abrir una ventana con la que podamos trabajar. Esta ventana será la del fondo, la madre de todas las subsiguientes ventanas y donde se alojará el menú principal del programa. La llamada es un método de la clase *window*, heredada por *application* y que veremos en la siguiente sección. El destructor de la clase contiene una única instrucción, *endwin()*, la función de curses utilizado para recoger la basura y para devolver la terminal al estado que tenía antes de la llamada a *initscr()*. De momento es todo lo que necesitaremos para salir elegantemente de *curses*.

Abriendo la Ventanas

En la siguiente capa de nuestra aplicación, estaría la clase que administraría cada una de las ventanas que se fuesen creando, incluyendo la principal. Esta capa viene representada por la clase *window*, la implementación de la cual se puede ver en el Listado 2 (o al menos parte. Se han dejado fuera los métodos *get* y *set* correspondientes a varios atributos. Ver [2]). Fijémonos que window cuenta con dos constructores sobrecargados. Uno, el que no tiene parámetros, sirve de constructor cuando se crea un objeto application y crea el entorno curses. Este constructor se llamará una sola vez por aplicación y monta la ventana stdscr, el "contenedor" de todas las demás ventanas. En el código se aprecia como usa los métodos estándar de inicializar la pantalla y librería de curses utilizando (*initscr()*) y como se activa el mapeado del teclado (*keypad()*, esto sirve para que las teclas de función, cursores, etc. devuelvan caracteres que el programa pueda procesar, permitiendo, por ejemplo, que si el usuario pulsa la tecla F1, se active el sistema de ayuda, etc.). A continuación, deshabilitamos la secuencia de Nueva Línea + Retorno de Carro cada vez que se produce una salida con la llamada a *nonl()* y le decimos a la aplicación que ha de capturar las pulsaciones en cuanto se produzcan sin esperar a un carácter de nueva línea con el procedimiento *cbreak()*. Esto último nos permitirá procesar cada tecleo del usuario como es debido, activando alguna funcionalidad del programa si pulsa un tecla de un carácter no imprimible o colocando una letra en la ventana apropiada si el usuario desea escribir algo. Por fin, evitamos que se visualicen inmediatamente las entradas desde el teclado con la función *noecho()*. De esta manera, podremos capturar los tecleos y procesarlos como nos convenga en el contexto de la aplicación. Todas estas funciones pertenecen a la librería de curses y su uso es bastante estándar en el arranque de cualquier aplicación que utilice el paquete. El segundo constructor, *window::window(int wide = COLS,int high = LINES, int posX = 0,int posY = 0, bool frame = TRUE, string win_Name = "Untitled"...)*, sirve para crear las ventanas con las que podremos interactuar. Si nos fijamos, el constructor de la clase *application* utiliza este constructor para generar el fondo visible de la aplicación, con un marco (establecido por el parámetro *frame*) y de un tamaño igual a la del terminal donde se mostrará. El tamaño de la terminal se establece en líneas (*LINES*) y columnas (*COLS*), siendo estas dos variables generadas por curses a la hora de inicializar el motor y para la ventana principal de la aplicación vamos a ocupar todo el terminal visible. Por ello, al invocar al constructor de la ventana de fondo de la aplicación en el constructor de *application*, utilizamos el alto y ancho máximo disponible al arrancar el programa. De hecho, lo primero que se hace en este segundo constructor es crear una ventana con la función *curses newwin()* y asignar el puntero que devuelve al atributo *w_Handle*. Este atributo nos será útil más adelante para referirnos a cada una de las ventanas cuando contemos con más de uno. A continuación, procesamos el título de la ventana de manera similar que hacíamos en el constructor de *application* y establecemos al atributo *frame* para dibujar (o no) un marco alrededor de la ventana. Lo siguiente es asignar la ventana a un panel. Los paneles en curses añaden propiedades a las ventanas permitiendo que estas se apilen por capas y se solapen de manera consistente. Para entendernos, sería muy difícil tener ventanas de diálogos, ventanas móviles y ventanas apiladas sin que estas participaran de las funcionalidades que les atribuyen los paneles. La conversión de una ventana "normal" en un ventana panel es tan sencilla como se ve en el listado: basta invocar a la función curses *new_panel()* con el handle de la ventana a convertir. La función devuelve un handle al panel creado que guardamos en el atributo *p_Handle* para referencias futuras. Luego llamamos al método de la clase *putTitle()* (que se ve un poco más abajo en el listado). Éste método, como su nombre indica, coloca el título de la ventana en el ángulo superior izquierdo y, de paso, dibuja el marco para la ventana. El siguiente método, *showPanels()*, también perteneciente a la clase *window*, muestra el panel llamando a *update_panels()* (que actualiza el aspecto del panel) y *doupdate()* (que actualiza la pantalla con la nueva información), ambas funciones pertenecientes a la librería curses. El método *putTitle()* a su vez invoca al método *wWrite()*, que es el método de la clase encargado gestionar el paso de cadenas a las ventanas de *curses*. Recibe tres o más parámetros: los dos primeros establecen la posición x e y de la cadena (tercer parámetro) que se montará junto con los parámetros indefinidos que le siguen. La función *mvwprintw()*, perteneciente a la

Listado 1: application.cpp

```
01 #include "application.h"
02
03 application::application(bool
   frame=TRUE,string
   app_Name="Untitled"...)
04 {
05     makeString(app_Name);
06     title=app_Name;
07
08     window
   app_window(COLS,LINES,0,0,TR➥
   UE,title);
09 }
10
11 application::~application()
12 {
13   endwin();
14 }
15
16 // ... Y 'gets' varios
```

033-036-curses 22.11.2004 17:48 Uhr Seite 35

Listado 2: window.cpp

```
01 #include "window.h"
02
03 window::window()
04 {
05    w_Handle=initscr();
06    keypad(stdscr, TRUE);
07    nonl();
08    cbreak();
09    noecho();
10 }
11
12 window::window(int wide=COLS,int high=LINES,int
   posX=0,int posY=0,bool frame=TRUE,string
   win_Name="Untitled"...)
13 {
14    w_Handle=newwin(high,wide,posY,posX);
15
16    makeString(win_Name);
17    title=win_Name;
18    has_Frame=frame;
19
20    p_Handle=new_panel(w_Handle);
21
22    putTitle();
23    showPanels();
24 }
25
26
27 void window::closeWindow()
28 {
29    delwin(w_Handle);
30    refresh();
31 }
32
33 void window::putTitle()
34 {
35    int px=1,py=0;
36    if(has_Frame)
37    {
38       drawFrame();
39       px=0;py=-1;
40    }
41    wWrite(px,py,title);
42 }
43
44 void window::showPanels()
45 {
46    update_panels();
47    doupdate();
48 }
49
50 void window::wWrite(int px, int py, string
   my_String...)
51 {
52    if (has_Frame)
53    {
54       px++;py++;
55    }
56    makeString(my_String);
57    mvwprintw(w_Handle,py,px,my_String.c_str());
58    wrefresh(w_Handle);
59    showWindow();
60 }
61
62 void window::showWindow()
63 {
64   nodelay(w_Handle,TRUE);
65   wgetch(w_Handle);
66   nodelay(w_Handle,FALSE);
67   showPanels();
68 }
69
70 void window::drawFrame()
71 {
72    wborder(w_Handle, ACS_VLINE, ACS_VLINE,
   ACS_HLINE, ACS_HLINE, ACS_ULCORNER, ACS_URCORNER,
   ACS_LLCORNER, ACS_LRCORNER);
73
74 }
75
76 int window::wGetch()
77 {
78    return(wgetch(w_Handle));
79 }
80
81 // Gets y sets varios aquí
```

librería curses, es la encargada de colocar la cadena en la ventana indicada por el handle que se le pasa. Nótese como, a pesar de que la mayoría nosotros estamos acostumbrados a referirnos primero al eje de las x y después al eje de las y, curses lo hace a la inversa, más bien porque cuando hablamos de terminales de texto, es habitual hablar primero de líneas (componente vertical) y después de columnas (componente horizontal) en ese orden. Después de colocar la cadena, llamamos *wrefresh()* que refresca la ventana que acabamos de modificar, que no la pantalla. El refresco de la pantalla para que se puedan ver las modificaciones se produce en el método *showWindow()*, que llamamos a continuación. El método *showWindow()* es un parche en la ya larga tradición de los famosos parches de Paul C. Brown. Nació cuando me dí cuenta de que *curses* no me actualizaba satisfactoriamente las ventanas si no había una entrada de datos desde el teclado. Bueno, más bien, si no se intentaba conseguir una entrada desde el teclado. Por ello lo único que hace esta rutina es anular la espera para una entrada *(nodelay())*, leer el búfer de entrada (*wgetch()* – normalmente dicho búfer estará vacío, puesto que no espera a que el usuario teclee nada) y volver a activar la espera de entrada para, a continuación, actualizar los paneles llamando a *showPanels()*. Sin este procedimiento, los paneles no se actualizan

033-036-curses 22.11.2004 17:48 Uhr Seite 36

hasta una nueva entrada por parte del usuario y no se mostrarán de motu propio cosa como los títulos o el borde. En cuanto a *drawFrame()*, su nombre lo dice todo: dibuja un marco alrededor de la ventana utilizando para ello la rutina curses *wborder()* y los valores predefinidos de curses para la barra vertical (*ACS_VLINE*), tanto para el borde izquierdo, como para el derecho; la barra horizontal (*ACS_HLINE*) para el borde superior e inferior ; y los gráficos de las esquinas *ACS_ULCORNER* (esquina superior izquierdo), *ACS_URCORNER* (esquina superior derecho), *ACS_LLCORNER* (esquina inferior izquierdo), *ACS_LRCORNER* (esquina inferior derecho). Por fin, el método *wGetch()* se encarga de recoger las entradas procedentes del teclado referidas a la ventana para que puedan ser procesadas.

Parámetros indefinidos

Conviene explicar en un aparte lo referente a las funciones con parámetros indefinidos. Estas funciones permiten que funcionen, por ejemplo, el socorrido *printf()* de C. Si nos fijamos en como funciona, si hacemos

```
printf("Tengo %i muñeca vestida de %s...",1,"azul");
```

devolverá la cadena "Tengo 1 muñeca vestida de azul..." y comprobamos que a *printf()*, que no es más que otra función de las librerías estándar de C, puede recibir tantos parámetros como gustemos... siempre y cuando el primero sea el que indique el formato del mensaje a mostrar. Eso es por que *printf()* es una función que acepta, además de una serie de parámetros definidos, un número indefinido de otros parámetros. Hay varias maneras de leer los parámetros indefinidos a variables. Una sería la de pasar como argumento definido el número de argumentos indefinidos. Otra la de incluir como definido un argumento que fuese igual al último de los argumentos indefinidos. En ambos casos crearíamos un bucle (en el primer caso un bucle *for* y en el segundo uno *while*) que fuera iterando sobre la lista de argumentos hasta que o (a) se leyese hasta el número de argumentos preestablecido o (b) se llegase hasta el argumento que fuese igual al argumento definido como el último. Ambos métodos ofenden mi sentido de la estética de la programación, ya que ¿qué pasa si ni el mismo programador sabe cuantos argumentos van se indefinidos? Yo opto por el método (c), que consiste en que todos los argumentos se vuelcan en una lista (de tipo *va_list*) y se van leyendo e integrando en el argumento plantilla. Esto no sería un problema si no fuera porque con curses, el uso de una cadena de longitud indefinida (tipo *char *cadena*) genera un fallo de segmentación. En concreto, si utilizamos una cadena de longitud indefinida **buf* en un programa C o C++ "normal" (es decir, que no cargue librería "raros" como curses) no pasa nada:

```
function ejemplo2(char *plant, ...)
{
 char *buf;
 va_list args;
 va_start(args, plant);
vsprintf(buf,plant,args);
 va_end(args);
 return(buf);
}
```

El código anterior funcionará sin ningún problema. Pero con curses, si se utiliza una cadena de longitud indeterminada y después se utiliza *sprintf()* o *vsprinf()* con una plantilla y varios argumentos, el resultado es el dichoso error de fallo de segmentación, uno de los errores más irritantes que tiene la insidiosa costumbre de aparecer en el momento de la ejecución, después de una compilación exitosa. Otra pega un tanto engorrosa es que *va_start()* y *va_end()* han de estar en la misma función que recibe los parámetros indefinidos, por tanto veréis la

Listado 3: va.h

```
01 #include <stdlib.h>
02
03 #define makeString(return_String)       \
04 {                                       \
05   char *fmt;                            \
06   if ((fmt=(char*)
   malloc(sizeof(char)*return_String.length()))==NULL)\
07      printf("ERROR: Memoria insuficiente");\
08   else                                  \
09   {                                     \
10      strcpy(fmt,return_String.c_str()); \
11      va_list args;                      \
12      char *p;                           \
13      int n, size=10;                    \
14      if((p = (char*) malloc(size)) == NULL)\
15        printf("ERROR: Memoria insuficiente");\
16      else                               \
17      {                                  \
18        while(1)                         \
19        {                                \
20          va_start(args,fmt);            \
21          n=vsnprintf(p,size,fmt,args);  \
22          va_end(args);                  \
23          if (n < size)                      \
24          {                              \
25            break;                       \
26          }                              \
27          else                           \
28          {                              \
29            size = n+1;                  \
30            if((p = (char *) realloc(p,size)) ==
   NULL)\
31            {                            \
32              printf("ERROR: Memoria Insuficiente");\
33              break;                     \
34            }                            \
35          }                              \
36        }                                \
37      }                                  \
38      return_String=p;                   \
39      free(p);                           \
40      free(fmt);                         \
41   }                                     \
42 }
```

misma secuencia de comandos repetida una y otra vez a lo largo de las clases que estamos analizando y no hay manera de separarlos en una función aparte. El tercer problema es que sólo se nos permite operar con cadenas tipo *char* *, lo que teóricamente nos despoja del privilegio de utilizar cadenas de tipo *string*, con la de ventajas y facilidades que aportan, caray. Todos y cada uno de estos inconvenientes se resuelven con el macro que se puede ver en el Listado 3. ¿Cómo funciona? Pues, como se puede observar en el listado, lo primero que hacemos es asignar espacio a una variable de tipo *char** llamado *fmt*, de hecho tanto espacio como caracteres contenga el argumento *string (return_String)*. A continuación copiamos el contenido de *return_String* a *fmt*. a continuación asignamos un poco de espacio (*size en bytes*) a otra variable *char** llamado *p*. Esta variable contendrá la cadena procesada. Seguidamente, creamos la cadena final con un límite de *size* bytes. La función *vsnprintf()* asigna tantos bytes a *p* como los indicados en *size*, por tanto, no es posible excedernos de la cantidad de memoria asignada a *p* y evitamos posibles fallos de segmentación. Ahora bien, si toda la cadena no cabe en esos *size* bytes, *vsnprintf()* devuelve el número de bytes que hubiera cabido. Es decir, si *n* resulta ser mayor que *size*, es que se ha truncado la cadena. En este caso lo que hemos de hacer es aumentar el valor de *size* hasta *n+1* (la longitud total más el crácter nulo */0*) y reasignar memoria por el nuevo valor a *p*. Es lo que se hace a continuación. Una vez que tenemos el tamaño adecuado, volvemos a asignar la cadena a *p* y podemos salir del bucle y asignar la cadena contenida en *p* a *return_String* y proceder a liberar la memoria asignada a *p* y a *fmt*. Este ingenioso truco viene de la página man de *vsnprintf()*, si bien ha sido adaptado para que por un extremo entre una variable tipo string sin los otros parámetros para formatear y por el otro salga una variable string con todos los parámetros dispuestos limpiamente en su interior. Toda la basura se recoge, la memoria reservada para las cadenas se libera, todo queda inmaculadamente limpio tras su ejecución. Este macro no produce una compilación limpia. A la hora de compilar con g++ genera una advertencia allá donde se emplee que reza:

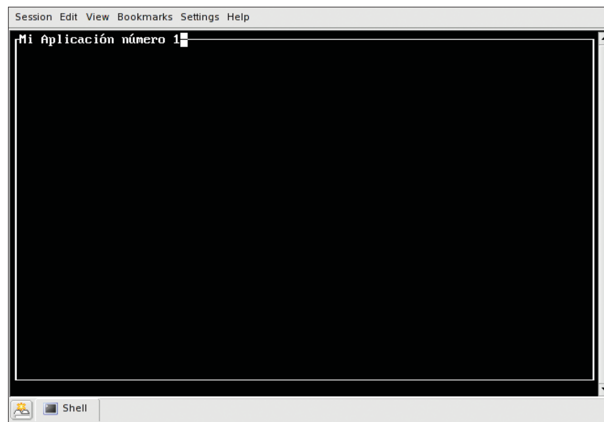

Figura 1: Nuestra primera aplicación, con marco y título.

```
warning: second parameter of
`va_start' not lastnamed
argument
```

Esta advertencia surge debido a que el compilador es incapaz de reconocer una cadena de tipo *string* como el último argumento definido antes de la ristra de argumentos indefinidos. Sin embargo, el programa acaba compilando y se ejecuta sin problemas.

Listado 4: principal.cpp

```
01 #include "application.h"
02
03 int main()
04 {
05    int ch;
06    application my_App(TRUE,"Mi
   Aplicación número %i",1);
07
08
   while((ch=my_App.wGetch())!=KE
   Y_HOME)
09    {
10
11    }
12
13    my_App.~application();
14    exit(0);
15 }
```

Explotación

Por fin hemos llegado al momento de ver nuestras clases en acción. Si miramos el Listado 4, vemos que la función *main* de nuestro programa es harto sencillo. Incluimos el fichero de inclusión application.h que contiene la definición de la clase *application*. A continuación, declaramos una variable tipo *char* para contener las pulsaciones del teclado y un objeto tipo *application*. Llegados a este punto, se visualizará en la terminal la ventana principal de la aplicación, tal y como se ve en la Figura 1. Seguidamente, entramos en un bucle a la espera de que el usuario pulse la tecla INICIO o HOME, lo que cierra el bucle, desencadenando la destrucción del objeto *application* y cerrando el programa.

Compilación

Para la compilación con *curses*, hemos de enlazar con la librería *ncurses*. Si además empleamos paneles para poder solapar ventanas, hemos de incluir la librería *panels* y, por supuesto, hemos de enlazar las clases que hemos creado. Para todo ello existe un sencillo Makefile que se puede descargar junto con el resto del código fuente de [2].

Conclusión

Si bien parece que es poco lo obtenido hasta ahora, hemos sentado las bases para una aplicación mucho más compleja. Podríamos tomar lo desarrollado como una plantilla para un programa funcional e ir insertando nuevas funcionalidades con un de mínimo esfuerzo. El mes que viene seguiremos utilizando curses para implementar una aplicación y veremos como crear ventanas modales de diálogo, como crear botones y otras funcionalidades que nos demostrarán que los interfaces de texto siguen estando vivitos y coleando ■

RECURSOS

[1] Curses en plataformas Windows: *http://www.funet.fi/pub/win-nt/curses/* *http://www.crystalcom.com/crs_swin.htm* *http://www.eunet.bg/simtel.net/msdos/screen.html*

[2] Las fuentes completas, con ficheros de inclusión, para el programa descrito en este artículo *http://www.linuxnewmedia.es/magazine/numero01/descargas/curses*