Fundamentals of Programming 1

Terminal Handling — *curses* Library

Arkadiusz Chrobot

Department of Computer Science

January 22, 2020

Outline

Introduction

The C language was created for the Unix operating system, which originally interacted with the user only via terminals consisting of the keyboard and the computer monitor (the video display). The monitors were using a text mode to convey information to the user. In the text mode only characters can be displayed. The resolution of the monitor's screen in such a mode is expressed in the number of characters that can be displayed simultaneously on the screen. The most commonly used resolution was the 80×25, which means 80 columns and 25 rows. In each row and column only one character can be displayed. Some of the monitors allowed using colors for the characters and their background. Contemporary computers display information on the screen in a graphics mode, but the text mode is still available. In some cases working with the text mode is more efficient than working with the graphics mode. Some features offered by the graphics mode applications, like the Graphical User Interface (GUI) can be also, to some extend, adapted to the text mode based programs. The *curses* library has been created for this purpose.

The *curses* Library

There are at least three versions of the *curses* library. The *ncurses* (*new curses*) library is available for Unix compatible operating systems, like Linux. The *pdcurses* library (*public domain curses*) is for the MS Windows operating systems family. The original *curses* library is created for the Unix system. Modern versions of this library support using a mouse device and elements known from the GUI, that are in the Unix terminology called *widgets*. An addition to the library that contains a set of widgets is called CDK (*Curses Development Kit*). In the lecture only the basics features of the *curses* library and its derivatives are presented. A description of more advanced elements of the libraries can be found, for example, in this website: http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/.

Initialization and Finalization

To use the *curses* library the program has to include the `curses.h` header file. The primary function for initializing the library is the `initscr()`. It takes no arguments and returns an address (a pointer to) a structure of the `WINDOW` type. This value is often ignored, but it is worth checking, if it's not `NULL`. If it is, then the library initialization has not been successful. Other functions associated with the library initialization are described in the next slide.

Initialization an Finalization

`echo()/noecho()`	Both functions take no argument and return an `int` value. The former enables displaying on the screen characters typed on the keyboard, the latter has the reverse effect.
`keypad()`	The function enables support for additional keys on the keyboard, like arrow keys and function keys. It returns an `int` value and takes two arguments — a pointer to the `WINDOW` structure (explained latter) and a `bool` value, that switches the support on/off.
`halfdelay()`	The function enables a keyboard handling mode in which each typed character is available to the program immediately. Moreover, the function takes as an argument a timeout value of the `int` type. It is the time expressed in tenths of a second when the keyboard reading functions are waiting for a user to press a key. The function returns an `int` value.
`curs_set()`	The function sets the size and shape of the cursor. It takes an `int` value as an argument. If it is `0` then the cursor will be invisible. If the argument is `1` then the cursor will be of the size of the underscore character. Finally, if the argument is `2` then the cursor will be a block occupying a single character place on the screen.

All functions from the table return values expressed by two constants: `OK` and `ERR`. The former indicates that the function has successfully completed its task, the latter that an exception has occurred.

Initialization and Finalization

The `endwin()` function finalizes the use of the *curses* library. It takes no arguments and returns the `OK` or `ERR` constants to indicate the status of finalizing the library.

Main Window

After the *curses* library is initialized with the use of the `initscr()` function, the `stdscr` global variable becomes available. The variable is a pointer to a structure of the `WINDOW` type associated with the screen. The type of the structure is defined in the library. Variables of this type describe properties of a single window. The `stdscr` variable points to a structure that describes the state of the main window, that covers the whole screen. Some of the library functions allow displaying characters, moving the cursor and many other operations concerning the main window. The starting point of the main window is the top left corner of the screen with $(0, 0)$ coordinates. The vertical coordinates grows "downwards" and the horizontal in the "left to right" fashion. The number of rows in determined by the `LINES` constant and the number of columns is determined by the `COLS` constant.

New Windows

The `newwin()` function creates new windows, which surfaces should be smaller or equal the size of the main window, and which don't "stick out" outside this window. The new windows mustn't overlap. If a overlapping windows are needed then an additional library named *panel* should be used. However, this library is not be discussed in this lecture. The `newwin()` function takes four arguments. The first one is the number of rows in the new window, the second one is the number of columns, the third and forth are coordinates of the point in the main window where the top left corner of the new window should be placed. **Please notice, that all the functions of *curses* library expect the coordinates to be passed to them in a reversed order – first the column number, then the row number.** The `newwin()` function returns the address of a `WINDOW` structure associated with the new window or `NULL`, if it fails to create the window.

Deleting a Window

The main window is removed by the `endwin()` function. The windows created with the use of the `newwin()` function are deleted by the `delwin()` function. It takes as an argument a pointer to the structure of the `WINDOW` type associated with the window to be deleted and it returns a value of the `int` type, which is interpreted in the same way as the value returned by other functions from the *curses* library (the `ERR` and `OK` constants).

Window Handling

With the help of the `mvwin()` function a window can be moved relatively to the main window. The function takes three arguments – a pointer to the structure of the `WINDOW` type associated with the window to be moved, and the new coordinates (y,x) of the top left corner of the window. Windows can be cleared with the use of the `erase()` and the `werase()` functions. The former clears the main window and doesn't take any arguments the latter clears a window associated with a structure pointed by the argument of the function. In the `curses` library terminology, a window is just an area of the screen and it's invisible. The simplest way of making it visible is to draw its edges with the use of `box()` function, which takes three arguments – a pointer to the structure of the `WINDOW` type associated with the window, a character which is used for drawing the horizontal edges of the window and a character that is used for drawing the vertical edges of the window. Those characters are specified by constants which names start with the `ACS_` prefix. However, if a zero is passed as the two last arguments of the function a default characters will be applied.

Refreshing a Window

With every window in the *curses* library is associated an area in the memory, which is called a *virtual window*. Any operation that changes the state of the window, changes the virtual window first. To make the changes visible on the screen, the window content has to be refreshed, or in other words, the changes have to be copied form the virtual window to the window on the screen. The `refresh()` function refreshes the main window, while the `wrefresh()` function refreshes any window that structure is pointed by the argument of the function. If the number of windows that requires refreshing is significant, it is better to call the `wnoutrefresh()` function for each of them and then invoke the `doupdate()` function only once. The former takes a pointer to a structure associated with the refreshed window as its argument and the latter takes no arguments. Each described function returns `OK` on success or `ERR` on failure.

Cursor Position

The `move()` function changes the cursor position in the main window. I takes new coordinates of the cursor as its arguments. The `wmove()` also changes position of the cursor but inside a specific window. It takes three arguments – a pointer to the structure associated with the window and the coordinates of the cursor inside the window. Those coordinates are relative to the top left corner of the window. Both function return one of the two constants: `OK` or `ERR`. To get the current coordinates of the cursor the `getyx` macro can be used. It takes three arguments – the pointer to the structure associated with a window, and two variables of the `int` type. The coordinates are store in the variables. If the macro is unsuccessful in obtaining them it stores `-1` in its arguments.

Displaying Text

The *curses* library provides some functions for displaying a single character or a string of characters on the screen. Aside from performing a similar operations to their counterparts declared in the `stdio.h` header file, some of them allow giving a specific attributes to the displayed characters like making them bold or printing them in italics. Each function from *curses* library that displays a text returns the `OK` on success or `ERR` in case of failure.

Displaying Single Characters

Several functions in the *curses* library are similar to the `putchar()` function from the standard C language library. The simplest ones are the `addch()` and `waddch()`. The former takes only one argument — the character that should be displayed in the main window. The character is actually displayed after the main window is refreshed and in the place on the screen where the cursor is located. The latter function performs a similar operation but in a window. An address of the structure associated with the window is passed to the function as its first argument. The second argument is the character. Both function allow specifying attributes of the character with the use of the *bitwise or* operator — `|`. The first argument of the operator is the character to be displayed and the second one is a constant specifying the attribute. There are many such constants defined in the *cuses* library, for example: `A_BOLD` — the character will be bold, `A_UNDERLINE` — the character will be underlined.

Displaying Single Characters

The `mvaddch()` and `mvwaddch()` functions perform similar operations to the functions described in the previous slide. They however take two additional arguments. The former function takes as a first two arguments the coordinates of the place in the main window where the character should be displayed. The latter function aside from the coordinates takes also, as the first argument, the structure associated with a window where the character should be displayed. All functions that display single characters can also display a special characters defined in the *curses* library by constants which names start with the `ACS_` prefix, for example: `ACS_BULLET` and `ACS_LARROW`. Those constants are also accepted as arguments by functions that display strings.

Displaying Strings

The *curses* library provides counterparts of the `puts()` function:

`addstr()`	The function takes a string as its argument. The string is displayed in the main window starting from the place where the cursor currently is and after the window is refreshed.
`addnstr()`	Performs similar operation to the previous function, but takes additional argument which is the maximum number of characters in the string that can be displayed.
`waddstr()`	Similar to the `addstr()` function but takes as the 1st argument an address of a structure associated with window where the string should be displayed.
`waddnstr()`	Similar to the `waddstr()` function but takes an additional, last argument which is the number of characters in the string to be displayed.
`mvaddstr()`	Similar to the `addstr()` function, but takes as its two first arguments the coordinates of the place in the main window from which the string should be displayed.
`mvaddnstr()`	Similar to the `mvaddstr()` function but takes an additional forth argument that limits the number of character of the string that should be displayed.
`mvwaddstr()`	Similar to the `mvaddstr()` function, but takes as an argument an address of a structure associated with a window where the string should be displayed.
`mvwaddnstr()`	Similar to `mvwaddstr()` function, but as the last argument takes the maximum number of characters in the string to be displayed.

Displaying Strings

The *curses* library also provides counterparts of the `printf()` function:

`printw()`	The function takes the same arguments as `printf()` and displays their values int the main window after it is refreshed and starting from the current position of cursor.
`wprintw()`	Similar to `printw()` but takes as the first argument a pointer to a structure associated with a window where the values should be displayed.
`mvprintw()`	Similar to `printw()` but as the two first arguments takes the coordinates of the place in the main window, from where the string should be displayed.
`mvwprintw()`	Similar to the `mvprintw()` but takes as the first argument the pointer to a structure associated with a window where the string should be displayed.

Attributes of Characters

The way of specifying attributes for single characters is explained in the previous slides. It can also be applied to describe attributes of strings of characters displayed by the `addstr()` and similar functions. However, the `printw()` and derivative functions require using separate functions for managing attributes. In case of the main window the `attrset()`, `attron()` and `attroff()` functions can be applied. All those functions return the `OK` constant on success or `ERR` on failure. Each of them takes also an `int` value as an argument. The value is usually defined as a constant and it is an attribute. The `attron()` function switches on an attribute. If it is invoked several times with different attributes, then all those attributes will be switched on. The `attrset()` function switches on a new attribute, but also switches off all attributes that have been already turned on. The `attroff()` function just switches off an attribute passed to it as an argument.

Attributes of Characters

There are counterparts of functions described in previous slide, which control attributes in any window. Their names start with the `w` letter and they take two arguments — a pointer to a structure associated with a window where the attributes have to be changed and an attribute defined usually by an appropriate constant.

Keyboard Handling

The *curses* library provides also functions for controlling and reading the keyboard input. The functions can read single characters or whole strings. The behaviour of those functions depends on the initialization of the keyboard handling by the *curses* library. For example if the `noecho()` function is called then the functions reading the input won't display the typed characters on the screen. However, if the `echo()` function is invoked, then those functions will print the input on the screen, but only after a window where it should be displayed is refreshed.

Reading Single Characters

Some versions of the *curses* library switch on by default a mode where the characters associated with pressed keys are available immediately to the program. Other apply the default mode used also by the functions from the standard C library — each key must be confirmed by the `Enter` key. The former mode can be switched on by invoking the `cbreak()` function, which takes no arguments and returns `OK` on success or `ERR` on failure. The standard mode of the keyboard handling can be switched on with the use of the `nocbrake()` function, which is invoked in the same way as the `cbreak()`. The `nodelay()` function can switch on or off the nonblocking mode of handling the keyboard. In nonblocking mode the program doesn't wait until the user presses a key. The function takes two arguments — a pointer to a structure associated with a window where the keyboard input should be printed and a value of the `bool` type, which specifies if the mode should be switched on of off. On success the `nodelay()` function returns `OK`. On failure it returns `ERR`. Another function that has an impact on the keyboard handling is the `halfdealy()` which is described in previous slides.

Reading Single Characters

Single characters can be read from the keyboard with the use of the `getch()` function. It returns the ASCII value of pressed key as an `int` value and takes no arguments. If the echo is switched on, the function also prints the read character in the main window at the current position of the cursor. If the nonblocking mode is turned on and the user hasn't pressed any key, then the function returns the `ERR` value. The `wgetch()` function is similar to the `getch()` function, but takes as an argument a pointer to a structure associated with a window where the character should be displayed if the echo is enabled. The `mvgetch()` function is also similar to `getch()`, but takes as arguments coordinates of the place in the main window, where the character should be displayed. The `mvwgetch()` function is similar to the `mvgetch()` function but as its first argument takes a pointer to a structure associated with a window where the character should be displayed if the echo is enabled.

Reading Single Characters

For the values of the special keys like function keys and arrow keys are defined constants in the *curses* library. For example the `KEY_LEFT` constant describes the value associated with the key that moves the cursor one position to the left. There is also a macro named `KEY_F` which takes as a argument the number of a function key and expands to its value. Those constants and the macro should be used only if handling the special keys is enabled.

Reading a String

The *curses* library provides also the counterparts of the `gets()` function:

`getstr()`	The function reads a string and stores it in an array of characters passed to the function as its argument. **Using the function can be dangerous because getstr() doesn't limit the number of entered characters. It's better to avoid using it.**
`getnstr()`	Similar to the `getstr()` function but takes an additional argument, which is the maximum number of characters that can be read from the keyboard. It's a safer replacement of the `getstr()` function.
`wgetstr()`	Similar to the `getstr()` function but prints the string from keyboard in a window which structure is pointed by its first argument. **It's better to avoid using it.**
`wgetnstr()`	The safer implementation of the `wgetstr()` function. It takes as the last argument the maximum number of characters that can be read from the keyboard.
`mvgetstr()`	Similar to the `getstr()` function. It takes as its two first arguments the coordinates of a place in the main window from which it should start printing the string read from the keyboard. **Avoid using it.**
`mvwgetstr()`	Similar to the `mvgetstr()` function, but takes as the first argument the pointer to structure associated with window where the string read from the keyboard should be printed. **Also avoid using it.**

Reading a String

`mvgetnstr()`	The safer version of the `mvgetstr()` function. The maximum number of characters that can be read from the keyboard is passed to it as its last argument.
`mvwgetnstr()`	The safer version of the `mvwgetstr()` function. The maximum number of characters that can be read from the keyboard is passed to it as its last argument.

All functions described in this and previous slide return `OK` on success and `ERR` in case of failure.

Reading a String

The *curses* library provides also the counterparts of the `scanf()` function:

`scanw()`	The function takes the same arguments as the `scanf()` function. It prints the keyboard input in main window, provided the echo is enabled.
`wscanw()`	Similar to the `scanw()` function, but it takes as its first argument a pointer to the structure associated with a window where the keyboard input should be displayed, provided the echo is enabled.
`mvscanw()`	Similar to the `scanw()` function, but as its two first arguments takes a coordinates of a place in the main window from which it should start printing the keyboard input, provided the echo is enabled.
`mvwscanw()`	Similar to the `mvscanw()` function, but takes as the first argument a pointer to a structure associated with a window in which the keyboard input should be displayed, provided the echo is enabled.

All the functions described in the table return `OK` on success or `ERR` on failure.

Colours Handling

Colours are attributes of displayed characters. Not all terminals allow using them. To check if the colours are available in a specific terminal the `has_colors()` function can be used. It returns a `bool` value. If it is `TRUE` then the terminal can display colours. After checking the availability of colours the `start_color()` function should be invoked. It takes no arguments and returns `OK` if it is able to initialize colours handling. Otherwise it returns `ERR`. The number of available colours is given by the `COLORS` constant. However, the *curses* library doesn't allow using a single colour. Pairs of colors have to be configured before colours can be applied. The first colour in the pair is the colour of a character, the second is the colour of the character's background. The maximum number of colour pairs is given by the `COLOR_PAIRS` constant. The `init_pair()` function configures a single pair of colours. It takes three arguments of the `short int` type. The first is the number of the pair and it should be greater than zero. The next two are the number of the character colour and the number of its background colour.

Colours Handling

The `init_pair()` function returns `OK` on success or `ERR` on failure. The `COLOR_PAIR` macro allows the programmer to chose a pair of colors. It takes as its argument the number of the pair and returns the colours in the pair as an attribute that can be assigned to displayed characters with, for example, the use of the `attron()` function. The table contains the list of constants that define colours and their description.

COLOR__BLACK	black colour
COLOR__RED	red colour
COLOR__GREEN	green colour
COLOR__YELLOW	yellow colour
COLOR__BLUE	blue colour
COLOR__MAGENTA	crimson-like colour
COLOR__CYAN	greenish-blue colour
COLOR__WHITE	white colour

Examples

The source code of all presented examples is available on the course web page. They are prepared to be used with the Code::Blocks programming environment. All of them, except for one are also presented in full in the slides. To preserver the legibility of the source code of programs the exception handling is reduced to the necessary minimum — if a function fails to complete its task then the program aborts. The better way of handling an exception would be to reset the terminal setting to their original values and then abort the program. The programs from the web page are configured to be used with the *ncurses* library. To make them work with the *pdcurses* library some modifications are necessary.

First Example — A Simple Program

```c
#include<curses.h>
#include<locale.h>

int main(void)
{
        if(setlocale(LC_ALL,"")==NULL)
                return -1;
        if(initscr()==NULL)
                return -1;
        printw("Hello, World!\n");
        if(refresh()==ERR)
                return -1;
        getch();
        if(endwin()==ERR)
                return -1;
        return 0;
}
```

Comment to the First Example

The program initializes the *curses* library, prints the famous "Hello, World!" sentence and waits until the user presses any key, then it finalizes the *curses* library and exits. The "Hello, World!" sentence is displayed after the main window is updated with the use of `refresh()` function. Waiting for the user to press any key is accomplished with the use of the `getch()` function. The `setlocale()` function used at the beginning of the program is declared in the `locale.h` header file and is used for setting the localization of the program. Since the program displays messages in English the function usage is optional.

Second Example — Reading Keys

```c
#include<curses.h>
#include<locale.h>

void print_keys(void)
{
        int key;
        do {
                key = getch();
                printw("The %c key was pressed.\n",key);
                refresh();
        } while(key!='q');
}
```

Second Example — Reading Keys

```
int main(void)
{
        if(setlocale(LC_ALL,"")==NULL)
                return -1;
        if(initscr()==NULL)
                return -1;
        if(noecho()==ERR)
                return -1;
        print_keys();
        if(endwin()==ERR)
                return -1;
        return 0;
}
```

Second Example — a Comment

In the `main()` function the *curses* library is initialized and the echo is disabled with the use of the `noecho()` function. This means that the `getch()` function is not displaying the characters associated with the keys it reads in the main window. The aforementioned function is invoked inside the `do…while` loop in the `print_keys()` function. The loop terminates after the user presses the `q` key. The information about pressed keys is displayed in the main window with the use of the `printw()` function. Please notice, that pressing some of the keys, like for example the arrow keys causes the program to print more than one character on the screen.

Third Example — Moving the Cursor

```
#include<curses.h>

void move_cursor(WINDOW *window)
{
    int x=0,y=0;

    getyx(window,y,x);
    int key = 0;
    do {
        key = getch();
        switch(key) {
            case KEY_LEFT:
                x=(x+(COLS-1))%COLS;
                move(y,x);
                break;
            case KEY_RIGHT:
                x=(x+1)%COLS;
```

Third Example — Moving the Cursor

```
            move(y,x);
            break;
        case KEY_UP:
            y=(y+(LINES-1))%LINES;
            move(y,x);
            break;
    case KEY_DOWN:
            y=(y+1)%LINES;
            move(y,x);
            break;
    case KEY_F(3):
            getyx(window,y,x);
            printw("x: %d, y: %d",x,y);
            break;
```

Third Example — Moving the Cursor

```
            case KEY_F(2):
                y=x=0;
                erase();
                break;
        }
        refresh();
    } while(key!='q');
}
```

Third Example — Moving the Cursor

```
int main(void)
{
    if(initscr()==NULL)
        return -1;
    if(keypad(stdscr,TRUE)==ERR)
        return -1;
    move_cursor(stdscr);
    if(endwin()==ERR)
        return -1;
    return 0;
}
```

Third Example — a Comment

The program allows the user to move the cursor around the whole screen with the use of the arrow keys. The *curses* library is initialized and finalized in the `main()` function. Also the `keypad()` function is invoked there. It initializes the handling of special keys by appropriate functions. The movement of the cursor is programmed in the `move_cursor()` function which takes as an argument the pointer to the structure associated with the main window. Inside the function, the `getyx` macro is used for getting the coordinates of the cursor current position. Those coordinates are stored in the `x` and `y` variables. Then the `key` variable is declared and initialized. The variable is used for storing a code of a pressed key returned by the `getch()` function. The latter function is invoked in the `do…while` loop. That code is recognized inside the `switch` statement. If its value is equal to the value of one of the `KEY_RIGHT`, `KEY_LEFT`, `KEY_UP` or `KEY_DOWN` constants then the cursor is moved accordingly by one place.

Third Example — a Comment

The movement consists in calculating coordinates of a new position of the cursor and invoking the `move()` function with those coordinates as its arguments. The modular arithmetics is used for calculating the coordinates of the next position of the cursor, so if the cursor "passes" one of the edges of the screen, it will appear on the other side of it. If the user presses the `F2` key, then the program will display coordinates of the previous position of the cursor. The coordinates are obtained with the use of the `getyx` macro and displayed with the use of the `printw()` function. Pressing the `F3` key causes the program to clear the main window by invoking the `erase()` function and to zero out the variables that store the coordinates of the cursor. Outside the `switch` statement but inside the `do…while` loop the `refresh()` function is called for updating the content of the main window. The loop terminates after the user presses the `q` key.

Forth Example — Windows

```
#include<curses.h>
#include<locale.h>

void move_window(WINDOW *window, int x, int y)
{
    int key=0;
    do {
        key = getch();
        if(key==' ') {
            x=(x+1)%10;
            y=(y+1)%10;
            erase();
            refresh();
            if(mvwin(window,y,x)==ERR)
                printw("Window out of the allowed area!\n");
            if(wrefresh(window)==ERR)
                printw("Window update failure!\n");
        }
    } while(key!='q');
}
```

Forth Example — Windows

```
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL)
        return -1;
    if(initscr()==NULL)
        return -1;
    if(curs_set(0)==ERR)
        return -1;
    WINDOW *window = newwin(5,10,0,0);
    if(window==NULL)
        return -1;
    if(box(window,0,0)==ERR)
        return -1;
    if(refresh()==ERR)
        return -1;
    if(wrefresh(window)==ERR)
        return -1;
    move_window(window,0,0);
    if(delwin(window)==ERR)
        return -1;
    if(endwin()==ERR)
        return -1;
    return 0;
}
```

Forth Example — a Comment

The program localization and initialization of the *curses* library is performed in the `main()` function. Then the cursor is made invisible with the use of the `curs_set()` function. Next, a window of the size 5×10 is created in the top left corner of the screen with the use of the `newwin()` function. Its edges are drawn with the use of the `box()` function. Then the program updates the content of the main window and the newly created window. Next, the `move_window()` function is invoked. The pointer to the structure associated with the newly created window is passed to the function, together with the coordinates of the left top corner of the window. Inside the function, in the `do…while` loop the ASCII code of a character associated with the pressed key is read with the use of the `getch()` function. If it is a space then coordinates of the new position of the top left corner of the window are calculated. Next the main window is cleared and the `mvwin()` function is invoked, that moves the newly created window.

Forth Example — a Comment

The program checks if the function completed its task correctly, although the terminal would have to have a very small resolution for making the window impossible to move. The window is visible in the new position after refreshing its content with the `wrefresh()` function. The result of invoking the function is also checked. The loop terminates after the user presses the `q` key. Before finalizing the *curses* library the program invokes the `delwin()` function to delete the window created by the `newwin()` function.

Fifth Example — Colours

```
#include<curses.h>
#include<locale.h>

void init_color_pairs(void)
{
    short int i,j, pair_counter=1;
    for(i=COLOR_BLACK;i<COLOR_WHITE;i++)
        for(j=COLOR_BLACK;j<COLOR_WHITE;j++) {
            if(init_pair(pair_counter,i,j)==ERR) {
                printw("Failed to initialize the %d pair of colours!\n",
                pair_counter);
                refresh();
            }
            pair_counter++;
        }
}
```

Fifth Example — Colours

```
void test_colors(void)
{
    short int i;
    for(i=1; i<COLOR_PAIRS; i++) {
        attron(COLOR_PAIR(i));
        printw("Test of the  %d pair of colours.\n",i);
        refresh();
        attroff(COLOR_PAIR(i));
        if(i%24==0) {
            getch();
            erase();
        }
    }
}
```

Fifth Example — Colours

```
int main(void)
{
    if(setlocale(LC_ALL,"")==NULL)
        return -1;
    if(initscr()==NULL)
        return -1;
    if(curs_set(0)==ERR)
        return -1;
    if(!has_colors())
        return -1;
    if(start_color()==ERR)
        return -1;
    init_color_pairs();
    printw("There are %d colours and  %d colours pairs.\n",COLORS,COLOR_PAIRS);
    refresh();
    getch();
    erase();
    test_colors();
    getch();
    if(endwin()==ERR)
        return -1;
    return 0;
}
```

Fifth Example — a Comment

The program localization and initialization of the *curses* library is performed in the `main()` function. Also the cursor is made invisible. Next, the program checks if colours are available in the terminal with the use of the `has_colors()` function. If so, then the colour handling is initialized with the use of the `start_color()` function and then pairs of colours are configured by the `init_color_pairs()` function which is defined in the program. It configures all possible colour pairs with the use of nested `for` loops. After the function completes its task the program informs the user about the number of available colours and pairs of colors and then waits until the user presses any key. The waiting is performed with the use of the `getch()` function. If the user presses a key then the main window is cleared and the `test_colors()` function is called.

Fifth Example — a Comment

Inside the `for` loop the `test_colors()` function switches on a pair of colours for a displayed text, with the use of `attron()` function and next print the text with the use of `printw()` and updates the contents of the main window by invoking the `refresh()` function. Next, it switches off the pair of colours with the use of the `attroff()` function. The `if` statement inside the loop causes the program to stop and wait for the user to press any key after each 24 printed lines. If the user presses the key, the program clears the main window. After the `test_colors()` function finishes the program finalizes the *curses* library and also exits.

Sixth Example — the Game of Life

The next example is a modified version of the game of life program that has been presented in the lecture on the multidimensional arrays. In this slides only the modified parts of the program source code are presented. The full version of this program is available on the course website. The only part of code that is changed and not presented here is the beginning, where the `stdio.h` header file is replaced by the `curses.h` header file.

Sixth Example — the Game of Life

```
char *error_msg[] = {
    "OK",
    "initscr() error",
    "noecho() error",
    "halfdealy() error",
    "start_color() error",
    "init_pair() error",
    "curs_set() error",
    "endwin() error"
};
```

Sixth Example — a Comment

The `error_msg` array contains messages that describe exceptions caused by *curses* library functions.

Sixth Example — the Game of Life

```
int initiate(void)
{
    if(!initscr())
        return -1;
    if(noecho()==ERR)
        return -2;
    if(halfdelay(2)==ERR)
        return -3;
    if(has_colors()!=FALSE) {
        if(start_color()==ERR)
            return -4;
        if(init_pair(1,COLOR_GREEN,COLOR_BLACK)==ERR ||
            init_pair(2,COLOR_BLACK,COLOR_BLACK)==ERR)
                return -5;
     }
     if(curs_set(0)==ERR)
        return -6;
     return 0;
}
```

Sixth Example — a Comment

The `initiate()` function is responsible for initializing the *curses* library. The function also switches off the echo and check colours availability. If there are available, then the color handling is enabled and two pairs of colours are initialized (green characters on a black background and black characters on a black background). The visibility of the cursor is turned off and the terminal is switched to a mode where the keyboard reading functions wait 0.2 seconds for the user to press any key. This mode is enabled with the use of the `halfdelay()` function.

Sixth Example — the Game of Life

```
WINDOW *create_board_window(void)
{
    int middle_y = LINES/2;
    int middle_x = COLS/2;
    int half_board = SIZE/2;
    int start_x = middle_x - SIZE;
    int start_y = middle_y - half_board;
    return newwin(SIZE,2*SIZE,start_y,start_x);
}
```

Sixth Example — a Comment

The `create_board_window()` function crates a window in the main window where the state of the game is displayed. First the coordinates of the middle point of the main window are calculated and stored in the `middle_x` and `middle_y` variables. Next, the half of the length of the board edge is calculated and stored in the `half_board` variable. Then, the coordinates of the top left corner of the board window are calculated and stored in the `start_x` and `start_y` variables. The width of the window is twice as big as the height, to make the board look on the screen as a square. It is necessary because the width of the columns is twice as small as the height of the rows. The `create_board_window()` function returns the address of a structure associated with the board window.

Sixth Example — the Game of Life

```
void print_board(unsigned char board[SIZE][SIZE], WINDOW *board_window)
{
    unsigned int i,j;

    for(i=0; i<SIZE; i++)
        for(j=0; j<SIZE; j++)
            if(board[i][j]) {
                (void)wattron(board_window,COLOR_PAIR(1));
                (void)mvwaddch(board_window,i,2*j,ACS_BULLET);
                (void)wattroff(board_window,COLOR_PAIR(1));
            } else {
                (void)wattron(board_window,COLOR_PAIR(2));
                (void)mvwaddch(board_window,i,2*j,' ');
                (void)wattroff(board_window,COLOR_PAIR(2));
            }
}
```

Sixth Example — a Comment

The `print_board()` function is responsible for displaying the state of the game which is stored in the matrix passed to the function by the `board` parameter. The pointer to a structure associated with the board window is passed to the function by its second parameter. Inside the nested `for` loops the value of every element of matrix is verified. If its value is not equal zero then it is displayed on the screen as a green dot (the `ACS_BULLET` constant) on a black background. Otherwise it is displayed as a black space on a black background. It won't be visible, but displaying it is necessary because in the previous iteration of the game the cell represented by the element could be alive and in the current iteration it should be dead, so its previous state should be overwritten by the current one. Please notice, that the horizontal coordinates of the places in the window are multiplied by two to compensate the ratio of the width to the hight of the window. The `void` keyword in parentheses before function calls means that the value returned by those functions is ignored.

Sixth Example — the Game of Life

```
int main(int argc, char **argv)
{
    int error = initiate();
    if(error!=0) {
        if(error<-1)
            (void)endwin();
        fprintf(stderr,"%s\n",error_msg[-error]);
        return -1;
    }
    WINDOW *board_window = create_board_window();
    if(!board_window) {
        printw("board_window() error\n");
        return -2;
    }
    if(argc==2) {
        if(!strcmp(argv[1],"blinker"))
            create_blinker(board);
        else if(!strcmp(argv[1],"ten_in_row"))
            create_ten_in_row(board);
        else
            seed_board(board);
    } else
        seed_board(board);
```

Sixth Example — the Game of Life

```
    while(getch()==ERR) {
        print_board(board,board_window);
        get_next_step(board);
        wrefresh(board_window);
    }

    if(delwin(board_window)==ERR) {
        printw("delwin() error\n");
        return -3;
    }

    if(endwin()==ERR) {
        fprintf(stderr,"%s\n",error_msg[7]);
        return -4;
    }

    return 0;
}
```

Sixth Example — a Comment

The `main()` function in this version of the program has been also modified. The `initiate()` function is called inside the `main()` function. If the former returns a value different than zero then a message about an exception associated with the initialization of *curses* library is displayed. If the returned value is less than -1 then the `endwin()` function is called and the program aborts. If the initialization is successful then a board window is created with the use of the `create_board_window()` function. If the function fails the program aborts. The `while` loop is also modified. In the body of the loop a modified version of the `print_board()` function and the `wrefresh()` function are called. In the header of the loop the condition is changed. The loop is performed until the user presses a key.

Sixth Example — a Comment

The state of the keys on the keyboard is inspected in the condition of the `while` loop with the use of the `getch()` function, which returns `ERR` until the user presses a key. The `halfdelay()` function called inside the `initiate()` function causes the `getch()` to wait 0.2 seconds for the user to press a key in each iteration of the `while` loop. Hence, the program displays subsequent states of the game with such a frequency. The user sees an animation of the state. Its speed can be changed by modifying the value of the `halfdelay()` function argument. After the loop finishes the board window is deleted with the use of `delwin()` function, the *curses* library is finalized and the program exits. The program runs only if the terminal it uses has at least 32 lines (the number of elements in a single dimension of the board). If the condition is not met, then it displays only a message about an exception.

Thanks

Many thanks to Grzegorz Łukawski, PhD, Leszek Ciopiński, MSc and Maciej Lasota, MSc for helping me to complete the Polish version of this slides.

Questions

?

The End

Thank You for Your attention!