12

12 Programs with functions and arrays

This chapter has a few examples that show some of the things that you can do using functions and arrays. A couple are small demonstration programs. Some, like the first example are slightly different. These examples– "curses", "menus", and "keywords" – entail the development of useful group of functions. These groups of functions can be used a bit like little private libraries in future examples.

12.1 CURSES

Curses?

Yes, well programs and curses are strongly associated but this is something different. On Unix, there is a library called "curses". It allows programs to produce crude "graphical" outputs using just low "cost cursor addressable" screens. These graphics are the same quality as the character based function plotting illustrated in 10.11.2. Unix's curses library is actually quite elaborate. It even allows you to fake a "multi-windowing" environment on a terminal screen. Our curses are less vehement, they provide just a package of useful output functions.

"Cursor addressable" screens and windows

These functions depend on our ability to treat the computer screen (or just a single window on the screen) as a "cursor addressable terminal screen". Basically, this means that this screen (or window) can be treated as a two-dimensional array of individually selectable positions for displaying characters (the array dimension will be up to 25 rows by 80 columns, usually a bit smaller). The run-time support library must provide a `gotoxy()` function and a `putcharacter()` function. Special character oriented input functions must also be used.

Run-time support in common IDEs

Such facilities are available through the run time support libraries provided with Symantec C++ for PowerPC, and Borland's IDE. In the Borland system, if you want cursor addressing facilities you have to create your project as either a "DOS Standard"

project, or an "EasyWin" project (there are slight restrictions on EasyWin and cursor graphics).

Naturally, the run-time support libraries in Symantec and Borland differ. So here we have to develop code that will contain conditional compilation directives so that appropriate code is produced for each different environment.

Separate header and implementation files

This code is to be in a separate file so that it can be used in different programs that require simple graphics output. Actually, there will be two files. Like the standard libraries, our curses "package" will consist of an implementation file with the definitions of the functions and a header file. The header file contains just the function prototypes and some constants. This header will be #included into the programs that need to use the curses display facilities.

We will need a small program to test the library. A "drawing program" would do. This could start by displaying a blank window with a "pen position" shown by a character such as a '*'. Keyed commands would allow the user to move the pen up, down, left and right; other commands could let the "ink" to be changed from a drawing pattern ('#') to the background ('.'). This would allow creation of pictures like that shown in Figure 12.1.

```
+----------------------------------------------------------------------------+
|............................................................................|
|............................................................................|
|............................................................................|
|............................................................................|
|.........####.....#......#..................................................|
|........#.........#......#.............##.....##............................|
|........#.......#####..#####...........##.....##............................|
|........#.........#......#..................................................|
|........#.........#......#..................................................|
|..........####.........................#.......#*...........................|
|........................................#######.............................|
|............................................................................|
|............................................................................|
|............................................................................|
|............................................................................|
|............................................................................|
|............................................................................|
|............................................................................|
+----------------------------------------------------------------------------+


>u
```

Figure 12.1 Picture produced using the cursor graphics package.

The IDE projects for programs built using the curses functions will have to identify the CG.cp ("Curses Graphics") file as one of the constituent files along with the normal main.cp. However, the header file, CG.h, will not be explicitly listed among the project files.

As noted earlier, the curses functions depend on a system's library that contains primitives like a `gotoxy()` function. IDEs normally have two groups of libraries – those that always get checked when linking a program, and those that are only checked when explicitly identified to the linker. In both the Symantec and Borland IDEs, the special run-time routines for cursor addressing are in files separate from the more common libraries. However, in both Symantec 8 and Borland, this extra library file is

among those checked automatically. In Symantec 7, the library with the cursor routines has to be explicitly made part of the project. If you do get linking errors when you try to build these programs, you will have to check the detailed documentation on your IDE to find how to add non-standard libraries to the set used by the linking loader.

Figure 12.2 illustrates how these programs will be constructed. A program using the curses routines will have at least two source files specified in its project; in the figure they are test.cp (which will contain `main()` and other functions) and CG.cp. Both these source files #include the CG.h routine. The CG.cp file also contains #include statements that get the header files defining the run-time support routines that it uses (indicated by the file console.h in the figure). The compiler will produce the linkable files test.o and CG.o. The linking loader combines these, and then adds all necessary routines from the normally scanned libraries and from any specifically identified additional libraries.

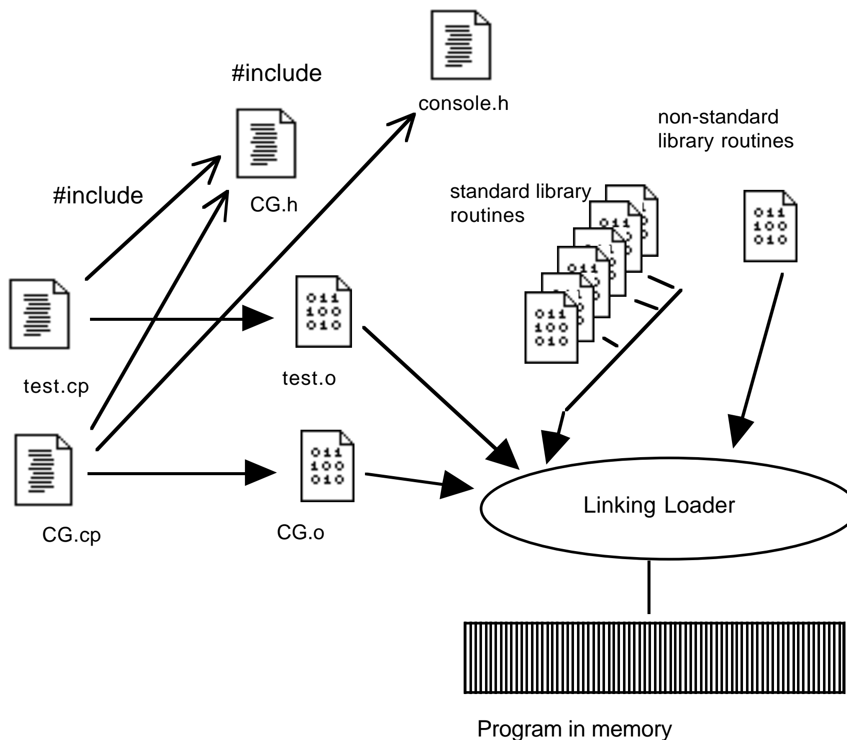


Figure 12.2 Creating a program from multiple files

Curses functions

The curses functions present the applications programmer with a simple window environment as shown in Figure 12.3. Characters are displayed by moving a "cursor" to a specific row and column position and then outputting a single character. If there are no intervening cursor movements, successive output characters will go in successive columns of the current row. (Output to the last column or last row of the screen should be avoided as it may cause unwanted scrolling of the display.)

As shown in Figure 12.3, the screen is normally divided into a work area (which may be marked out by some perimeter border characters), and a few rows at the bottom of the screen. These last few rows are used for things such as prompt line and data input area.

Input is usually restricted to reading single characters. Normally, an operating system will collect all characters input until a newline is entered; only then can the data be read by a program. This is not usually what one wants in an interactive program. So, instead, the operating system is told to return individual characters as they are entered.

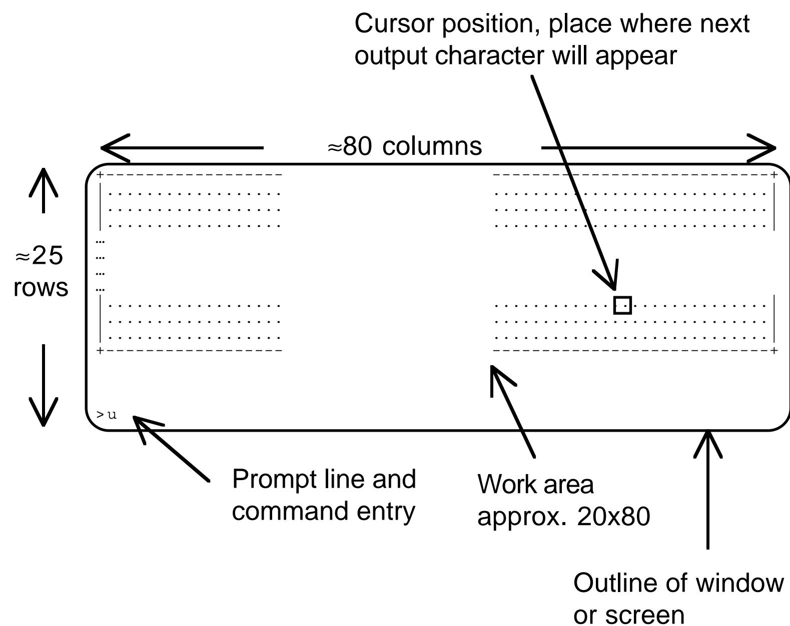


Figure 12.3 Curses model for an output window.

Often interactive programs need to display data that change. A program may need to arrange to pause for a bit so that the user can interpret one lot of data on the screen

before the display gets changed to show something else. Consequently, curses packages generally include a `delay()` function.

Minimal functionality required in a curses package

A curses package will have to provide at least the following:

1 Details of the width and height of the effective display area.

2 An initialize function.

In many environments, special requests have to be made to the operating system to change the way in which the terminal is controlled. These requests should go in an initialize function.

3 A reset function.

This would make the equivalent request to the operating system to put the terminal back into normal mode.

4 A clear window function.

This will clear all rows and columns of the display area, setting them to a chosen "background" character.

5 A frame window function.

Clears the window, then fills in the perimeter (as in the example shown in Figure 12.1).

6 A move cursor function.

This will use the IDE's `gotoxy()` function to select the screen position for the next character drawn ("positioning the cursor").

7 A put character function

Output a chosen character at the cursor's current position.

8 A prompt function that outputs a string on the prompt line.

9 A get character routine that returns the next character entered.

10 A delay function that can cause a delay of a specified number of seconds.

Prototypes of these functions are defined in the CG.h header file:

CG.h

```
#ifndef __CGSTUFF__
#define __CGSTUFF__
```

```
/*
Routines for a "cursor graphics" package.

These routines provide a consistent interface, there are
differing implementations for Intel (Borland) and PPC
(Symantec)
platforms.

This cursor graphics package is minimalist, it doesn't attempt
any optimizations.

The windows available for cursor graphics are usually 25 lines
by 80 columns.  Since output to the last line or last column
can sometimes cause undesired scrolling, a smaller work area is
defined.
*/

const int CG_WIDTH = 78;
const int CG_HEIGHT = 20;
const int CG_PROMPTLINE = 24;

/*
Initialize --- call before using cursor screen.
Reset --- call when finished.
*/

void CG_Initialize();
void CG_Reset();

/*
Movement and output
*/
void CG_MoveCursor(int x, int y);
void CG_PutCharacter(char ch);
void CG_PutCharacter(char ch, int x, int y);
void CG_ClearWindow(char background = ' ');
void CG_FrameWindow(char background = ' ');

/*
Delay for graphics effects
*/
void CG_Delay(int seconds);

/*
Prompting and getting input
*/
void CG_Prompt(const char prompt[]);
char CG_GetChar();

#endif
```

This file organization illustrates "good style" for a package of related routines. Firstly, the entire contents a bracketed by a conditional compilation directive:

Directives to protect against "multiple include" errors

```
#ifndef __CGSTUFF__
#define __CGSTUFF__

…
…
#endif
```

This mechanism is used to avoid compilation errors if the same header file gets #included more than once. Before the file is read, the compile time "variable" __CGSTUFF__ will be undefined, so #ifndef ("if not defined") is true and the rest of the file is processed. The first operation defines the "compile time variable" __CGSTUFF__; if the file is read again, the contents are skipped. If you don't have such checks, then multiple inclusions of the file lead to compilation errors relating to redefinition of constants like CG_WIDTH. (In larger programs, it is easy to get headers multiply included because header files may themselves have #include statements and so a particular file may get referenced in several places.)

Naming conventions

A second stylistic feature that you should follow is illustrated by the naming of constants and functions. All constants and functions start with the character sequence CG_; this identifies them as belonging to this package. Naming conventions such as this make it much easier for people working with large programs that are built from many separate source files. When they see a function called in some code, the function name indicates the "package" where that it is defined.

The functions are provided by this curses package are grouped so as to make it easier for a prospective user to get an idea of what the package provides. Where appropriate, default argument values are defined.

Run time support from IDEs

Run-time support functions of the IDEs

The different development environments provide slightly different versions of the same low-level support functions. Although the functions are provided, there is little in the IDEs' documentation to indicate how they should be used. The Borland Help system and manuals do at least provide full details of the individual functions.

In the Borland environment, the header file conio.h contains prototypes for a number of functions including:

```
int getche(void);              // get and echo character
void gotoxy(int x,int y);      // position cursor
int putch(int ch);             // put character
```

These specialized calls work directly with the DOS operating system and do not require any special changes to "modes" for keyboards or terminals. Borland "DOS/Standard" applications can use a `sleep()` function defined in dos.h.

In the Symantec environment, the header file console.h contains a prototype for

```
void cgotoxy(int x, int y, FILE *);     // position cursor
```

(the `FILE*` argument actually is set to a system provided variable that identifies the window used for output by a Symantec program). The prototype for the `sleep()` function is in unix.h, and the file stdio.h contains prototypes for some other functions for getting (`fgetc()`) and putting (`fputc()` and `fflush()`) individual characters. The handling of input requires special initializing calls to switch the input mode so that characters can be read immediately.

Delays in programs

The best way to "pause" a program to allow a user to see some output is to make a call to a system provided function. Following Unix, most operating systems provide a `sleep()` call that suspends a program for a specified number of seconds; while the program is suspended, the operating system runs other programs or deals with background housekeeping tasks. Sometimes, such a system call is not available; the Borland "EasyWin" environment is one such case. EasyWin programs are not permitted to use the DOS `sleep()` function.

If you need a delay in a program and don't have a `sleep()` function, or you need a delay shorter than one second, then you have a couple of alternatives. Both keep the program busy using the CPU for a specified time.

The more general approach is to use a compute loop:

```
void delay(int seconds)
{
    const long fudgefactor = 5000;
    double x;
    long lim = seconds*fudgefactor;
    while(lim>0) {
            x = 1.0/lim;
            lim--;
            }
}
```

The idea is that the calculation involving floating point division will take the CPU some microseconds; so a loop with thousands of divisions will take a measurable time. The `fudgefactor` is then adjusted empirically until appropriate delays are obtained.

There are a couple of problems. You have to change the `fudgefactor` when you move to a machine with a different CPU speed. You may get caught by an optimising compiler. A good optimising compiler will note that the value of `x` in the above code is never used; so it will eliminate the assignment to `x`. Then it will note that the loop has essentially no body. So it eliminates the loop. The assignment to `lim` can then be omitted; allowing the optimising compiler to reduce the function to `void delay(int)`

{ } which doesn't have quite the same properties. The compilers you usually use are much less aggressive about optimizing code so computational loops often work.

An alternative is to use system functions like `TickCounter()` (see 10.10). If your system has `TickCounter()` function that returns "seconds since machine switched on", you can achieve a delay using code like:

```
void delay(int seconds)
{
    long lim = TickCounter() + seconds;
    while(TickCounter() < lim)
           ;
}
```

(The function call in the loop will inhibit an optimising compiler, it won't try to change this code). Most compilers will give a warning about the empty body in the while loop; but it is exactly what we would need here.

Implementation of curses functions

The code to handle the cursor graphics functions is simple; it is largely comprised of calls to the run-time support functions. This code makes fairly heavy use of conditional compilation directives that select the specific statements required. The choice amongst alternatives is made by #defining one of the compiler time constants SYMANTEC, DOS, or EASYWIN.

The first part of the file #includes the appropriate system header files:

First part of CG.cp, #including selected headers

```
/*
Implementation of Cursor Graphics for Symantec 8
and Borland (either DOS-standard or EasyWin)

Versions compiled are controlled by compile time #defines
    SYMANTEC      for Mac/PC
    DOS
    EASYWIN
*/

#define SYMANTEC

#if defined(SYMANTEC)
/*
stdio is needed for fputc etc;
console  is Symantec's set of functions like gotoxy
unix for sleep function
*/
#include <stdio.h>
#include <console.h>
#include <unix.h>
```

```
#endif

#if defined(DOS)
/*
conio has Borland's cursor graphics primitives
dos needed for sleep function
*/
#include <conio.h>
#include <dos.h>
#endif

#if defined(EASYWIN)
/*
Still need conio, but can't use sleep, achieve delay by
alternative mechanism
*/
#include <conio.h>
#endif

#include "CG.h"
```

All version need to include CG.h with its definitions of the constants that determine the allowed width and height of the display area.

The various functions in the package are then defined. In some versions, the body of a function may be empty.

File CG.cp, Initialize() and Reset() functions

```
void CG_Initialize()
{
#if defined(SYMANTEC)
/*
Have to change the "mode" for the 'console' screen.
Putting it in C_CBREAK allows characters to be read one by one
as they are typed
*/
    csetmode(C_CBREAK, stdin);
#else
/*
No special initializations are needed for Borland environments
*/
#endif
}

void CG_Reset()
{
#if defined(SYMANTEC)
    csetmode(C_ECHO, stdin);
#endif
}
```

The `MoveCursor()` function makes the call to the appropriate run-time support routine. Note that you cannot predict the behaviour of the run-time routine if you try to move the cursor outside of the screen area! The run-time routine might well crash the system. Consequently, this `MoveCursor()` function has to constrain the arguments to fit within the allowed range:

File CG.cp: MoveCursor() function

```
void CG_MoveCursor(int x, int y)
{
    x = (x < 1) ? 1 : x;
    x = (x > CG_WIDTH) ? CG_WIDTH : x;

    y = (y < 1) ? 1 : y;
    y = (y > CG_HEIGHT) ? CG_HEIGHT : y;
#if defined(SYMANTEC)
    cgotoxy(x,y,stdout);
#else
    gotoxy(x,y);
#endif
}
```

Function `PutCharacter()` is "overloaded". Two versions are provided with slightly different argument lists. The first outputs a character at the current position. The second does a move to an explicit position before outputting the character. Note that the second function calls the more primitive `MoveCursor()` and `PutCharacter(char)` routines and does not duplicate any of their code. This is deliberate. If it is necessary to change some detail of movement or character output, the change need be made at only one place in the code.

File CG.cp, PutCharacter() functions

```
void CG_PutCharacter(char ch)
{
#if defined(SYMANTEC)
    fputc(ch, stdout);
    fflush(stdout);
#elif
    putch(ch);
#endif
}

void CG_PutCharacter(char ch, int x, int y)
{
    CG_MoveCursor(x,y);
    CG_PutCharacter(ch);
}
```

The `ClearWindow()` function uses a double loop to fill put "background" characters" at each position. (The run-time support routines provided by the IDEs may include additional "clear to end of screen" and "clear to end of line" functions that might be quicker if the background character is ' ').

The `FrameWindow()` function uses `ClearWindow()` and then has loops to draw top, bottom, left and right edges and individual output statements to place the four corner points.

File CG.cp ClearWindow() and FrameWindow() functions

```
void CG_ClearWindow(char background)
{
    for(int y=1;y<=CG_HEIGHT;y++)
            for(int x=1; x<=CG_WIDTH;x++)
                    CG_PutCharacter(background,x,y);
}

void CG_FrameWindow(char background)
{
    CG_ClearWindow(background);
    for(int x=2; x<CG_WIDTH; x++) {
                    CG_PutCharacter('-',x,1);
                    CG_PutCharacter('-',x,CG_HEIGHT);
            }
    for(int y=2; y < CG_HEIGHT; y++) {
                    CG_PutCharacter('|',1,y);
                    CG_PutCharacter('|',CG_WIDTH,y);
            }
    CG_PutCharacter('+',1,1);
    CG_PutCharacter('+',1,CG_HEIGHT);
    CG_PutCharacter('+',CG_WIDTH,1);
    CG_PutCharacter('+',CG_WIDTH,CG_HEIGHT);

}
```

The `Delay()` function can use the system `sleep()` call if available; otherwise it must use something like a computational loop:

File CG.cp, Delay() function

```
void CG_Delay(int seconds)
{
#if defined(EASYWIN)
/*
The EasyWin environment does not allow use of dos's sleep()
function.
So here do a "computational delay"
The value 5000 will have to be adjusted to suit machine
*/
    const long fudgefactor = 5000;
    double x;
    long lim = seconds*fudgefactor;
    while(lim>0) {
            x = 1.0/lim;
            lim--;
            }

#else
```

```
    sleep(seconds);
#endif
}
```

Function `Prompt()` outputs a string at the defined prompt position. This code uses a loop to print successive characters; your IDE's run time routines may include a "put string" function that might be slightly more efficient.

The `GetChar()` routine uses a run time support routine to read a single character from the keyboard.

File CG.cp, Prompt() and GetChar() functions

```
void CG_Prompt(const char prompt[])
{
#if defined(SYMANTEC)
    cgotoxy(1, CG_PROMPTLINE,stdout);
#else
    gotoxy(1, CG_PROMPTLINE);
#endif
    for(int i=0; prompt[i] != '\0'; i++)
            CG_PutCharacter(prompt[i]);
}

char CG_GetChar()
{
#if defined(SYMANTEC)
    return fgetc(stdin);
#elif
    return getche();
#endif
}
```

Example test program

<u>Specification</u>

The program to test the curses package will:

1 Display a window with a "pen" that the user can move by keyed commands.

2 The initial display is to show a "framed window" with '.' as a background character and the pen (indicated by a '*') located at point 10, 10 in the window. A '>' prompt symbol should be displayed on the promptline.

3 The pen can either be in "draw mode" or "erase mode". In "draw mode" it is to leave a trail of '#' characters as it is moved. In "erase mode" it leaves background '.' characters. Initially, the pen is in "draw mode".

4 The program is to loop reading single character commands entered at the keyboard. The commands are:

q or Q	terminate the loop and exit from the program
u or U	move the pen up one row
d or D	move the pen down one row
l or L	move the pen left one column
r or R	move the pen right one column
e or E	switch pen to erase mode
i or I	switch pen to ink mode

Any other character input is to be ignored.

5 The pen movement commands are to be restricted so that the pen does not move onto the window border.

Design

First iteration

There is nothing much to this one. The program structure will be something like:

```
move pen
    output ink or background symbol at current pen position
    update pen position, subject to restrictions
    output pen character, '*', at new pen position

main
    initialize
    loop until quit command
           get command character
           switch to select
                  change of pen mode
                  movements
    reset
```

The different movements all require similar processing. Rather than repeat the code for each, a function should be used. This function needs to update the x, y coordinate of the pen position according to delta-x and delta-y values specified as arguments. The various movement cases in the switch will call the "move pen" function with different delta arguments.

Second iteration

The pen mode can be handled by defining the "ink" that the pen uses. If it is in "erase mode", the ink will be a '.' character; in "draw mode" the character will be '#'.

The variables that define the x, y position and the ink have to be used in both `main()` and "move pen"; the x and y coordinates are updated by both routines. The x, y values could be made filescope – and therefore accessible to both routines. Alternatively, they

could be local to `main()` but passed by reference in the call to "move pen"; the pass by reference would allow them to be updated.

The only filescope data needed will be constants defining limits on movement and the various characters to be used for background, pen etc. The drawing limit constants will be defined in terms of the window-limit constants in the #included CG.h file.

File CG.h would be the only header file that would need to be included.

Third iteration

The loop structure and switch statement in main() would need a little more planning before coding. The loop could be a `while` (or a `for`) with a termination test that checks an integer (really a `boolean` but we don't have those yet in most C++ implementations) variable. This variable would initially be false (0), and would get set to true (1) when a 'quit' command was entered.

The switch should be straightforward. The branches for the movement commands would all contain just calls to the move pen function, but with differing delta x and delta y arguments.

Function `MovePen()` will have the prototype:

```
void MovePen(int& x, int& y, int dx, int dy, char ink)
```

The x, y coordinates are "input/output" ("value/result") arguments because the current values are used and then updated. These must therefore be passed by reference. The other arguments are "input" only, and are therefore passed by value.

<u>Implementation</u>

The file test.cp starts with its one #include (no need for the usual #include <iostream.h> etc). The CG files (CG.h and CG.cp) will have to be in the same directory as the test program. This doesn't cause any problems in the Borland environment as you can have a project folder with several different target programs that share files. In the Symantec environment, you may find it necessary to copy the CG files into the separate folders associated with each project.

After the #include, the constants can be defined:

```
#include "CG.h"

const int XMIN = 2;
const int XMAX = CG_WIDTH-1;
const int YMIN = 2;
const int YMAX = CG_HEIGHT-1;

const char pensym = '*';
const char background = '.';
const char drawsym = '#';
```

Function `MovePen()` is straightforward:

```
void MovePen(int& x, int& y, int dx, int dy, char ink)
{
    CG_PutCharacter(ink, x, y);

    x += dx;
    x = (x >= XMIN) ? x : XMIN;
    x = (x <= XMAX) ? x : XMAX;

    y += dy;
    y = (y >= YMIN) ? y : YMIN;
    y = (y <= YMAX) ? y : YMAX;

    CG_PutCharacter(pensym,x,y);
}
```

Function `main()` begins with declarations and the initialization steps:

```
int main()
{

    char ink = drawsym;

    int x = 10;
    int y = 10;

    int done  = 0;

    CG_Initialize();
    CG_FrameWindow('.');
    CG_PutCharacter(pensym,x,y);
```

Here, the loop is done using `for(;!done;)`; a `while` loop might be more natural:

```
    for(;!done;) {
           CG_Prompt(">");
           int ch;
           ch = CG_GetChar();
```

The switch handles the various possible commands as per specification:

```
           switch(ch) {
case 'i':
case 'I':
                  // put pen in drawing mode, the default
                  ink = drawsym;
                  break;
case 'e':
case 'E':
                  // put pen in erase mode
```

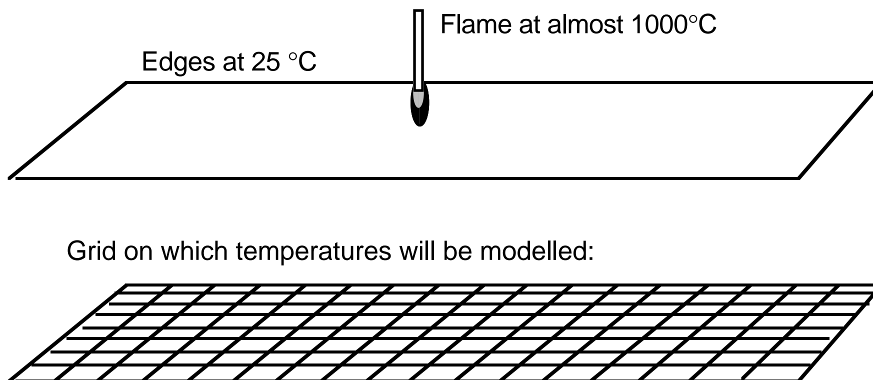
```
                    ink = background;
                    break;
case 'u':
case 'U':
                    MovePen(x, y, 0, -1, ink);
                    break;
case 'd':
case 'D':
                    MovePen(x, y, 0, 1, ink);
                    break;
case 'l':
case 'L':
                    MovePen(x, y, -1, 0, ink);
                    break;
case 'r':
case 'R':
                    MovePen(x, y, 1, 0, ink);
                    break;
case 'q':
case 'Q':
                    done = 1;
                    break;
default:
                    break;
                    }


            }
    CG_Reset();
    return 0;
}
```

12.2 HEAT DIFFUSION

Back around section 4.2.2, we left an engineer wanting to model heat diffusion in a beam. We can now do it, or at least fake it. The engineer didn't specify the heat diffusion formula so we'll just have to invent something plausible, get the program to work, and then give it to him to code up the correct formula.

Figure 12.4 illustrates the system the engineer wanted to study. A flame is to heat the mid point of a steel beam (of undefined thickness), the edges of which are held at ambient temperature. The heat diffusion is to be studied as a two dimensional system. The study uses a grid of rectangles representing areas of the beam, the temperatures in each of these rectangles are to be estimated. The grid can obviously be represented in the program by two dimensional array.

The system model assumes that the flame is turned on and immediately raises the temperature of the centre point to flame temperature. This centre point remains at this temperature for the duration of the experiment.




```
typedef double Grid[51][15];
```

Figure 12.4 The heat diffusion experiment.

Thermal conduction soon raises the temperature of the eight immediately neighboring grid square. In turn they warm their neighbors. Gradually the centre point gets to be surrounded by a hot ring, which in turn is surrounded by a warm ring. Eventually, the system comes to equilibrium.

The program is to display these changes in temperature as the system evolves. The temperatures at the grid squares are to be plotted using different characters to represent different temperatures ranges. Thus, '#' could represent temperatures in excess of 900°C, '@' for temperatures from 800 to 899°C and so forth. Figure 12.5 illustrates such plots.

The model for the heat diffusion is simple. (But it doesn't represent the real physics!) An iterative process models the changes of temperatures in unit time intervals. The temperature at each grid point increases or decreases so that is closer to the average of the surrounding eight grid points. For example, consider a point adjacent to the spot that is being heated, just after the flame is turned on:

```
25   25   25   25   25   …
25   25   25   25   25   …
25   25 1000   25 …
25   25   25   25 …
```

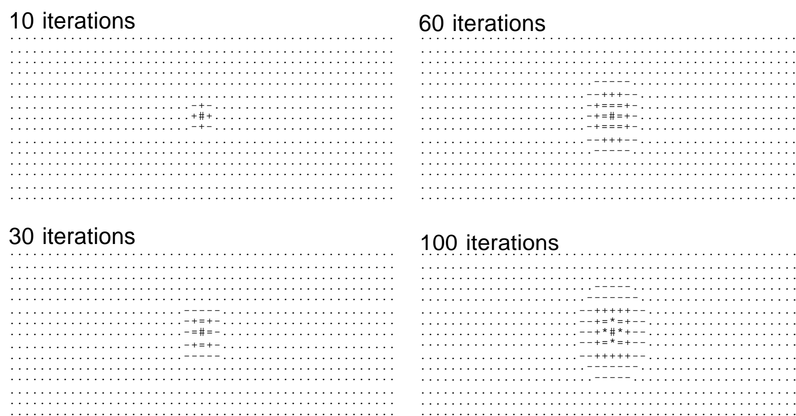


Figure 12.5 Temperature plots for the heat diffusion example.

The average of its eight neighbors is 146.8 ((1000 + 7*125)/8). So, this points temperature should increase toward this average. The rate at which it approaches the average depends on the thermal conductivity of the material; it can be defined by a formula like:

```
new_temp = conduct * average + (1 - conduct)* current
```

with the value of conduct being a fraction, e.g. 0.33. If the formula is defined like this, the new temperature for the marked point would be 65.1. This results in the temperature distribution

```
…
   25   25   25   25   25   …
   25   65   65   65   25  …
   25   65 1000   65 …
   25   65   65   65 …
…
```

(The centre point is held at 1000°C by the flame.) On the next iteration, the average temperature of this point's neighbors would be 156.8, and its temperature would increase to 95°C.

This process has to be done for all points on the grid. Note that this will require two copies of the grid data. One will hold the existing temperatures so that averages can be computed; the second is filled in with the new values. You can't do this on the same grid because if you did then when you updated one point you would change inappropriately the environment seen by the next point.

Once the values have been completed, they can be displayed. Changes in single iterations are small, so normally plots are needed after a number iterative cycles have

been completed. The "plotting" could use the cursor graphics package just implemented or standard stream output. The plot simply requires a double loop to print the values in each grid point. The values can be converted into characters by dividing the range (1000 - 25) into ten intervals, finding the interval containing a given temperature and using a plotting character selected to represent that temperature interval.

Specification

The program to model a two dimensional heat diffusion experiment:

1 The program will model the process using a two dimensional array whose bounds are defined by constants. Other constants in the program will specify the ambient and flame temperatures, and a factor that determines the "thermal conductivity" of the material.

2 The program will prompt the user for the number of iterations to be performed and the frequency of displays.

3 Once modelling starts, the temperature of the centre point is held at the "flame temperature", while the perimeter of the surface is held at ambient temperature.

4 In each iteration, the temperatures at each grid point (i.e. each array element) will be updated using the formulae described in the problem introduction.

5 Displays will show the temperature at the grid points using different characters to represent each of ten possible temperature ranges.

Design

First iteration

The program structure will be something like:

```
main
    get iteration limit
    get print frequency
    initialize grid
    loop
           update values in grid to reflect heat diffusion
           if time to print, show current values
```

Second iteration

Naturally, this breaks down into subroutines. Function `Initialize()` is obvious, as are `HeatDiffuse()` and `Show()`.

```
Initialize
```

```
        double loop sets all temperatures to "ambient"
        temperature of centre point set to "flame"

    Show
        double loop
        for each row do
               for each column do
                      get temperature code character
                             for this grid point
                      output character
               newline

    HeatDiffuse
        get copy of grid values
        double loop
        for each row do
               for each col do
                      using copy work out average temp.
                             of environment of grid pt.
                      calculate new temp based on current
                             and environment
                      store in grid
        reset centre point to flame temperature
```

While `Initialize()` is codeable, both `Show()` and `HeatDiffuse()` require additional auxiliary functions. Operations like "get temperature code character" are too complex to expand in line.

Third iteration

Additional functions have to be defined to handle these operations:

```
    CodeTemperature
        // find range that includes temperature to be coded
        set value to represent ambient + 0.1 * range
        index = 0;
        while value < temp
               index++, value += 0.1*range
        use index to access array of characters representing
               the different temperature ranges

    Copy grid
        double loop filling copy from original

    New temperature
        formula given  conductivity*environment + (1- cond)*current

    Average of neighbors
        find average of temperatures in three neighbors in row
               above, three neighbors in row below, and left, right
               neighbors
```

While most of these functions are fairly straightforward, the "average of neighbors" does present problems. For grid point r, c, you want:

```
g[r-1][c-1] + g[r-1][c]+ g[r-1][c+1] +
       g[r][c-1] + g[r][c+1] +
              g[r+1][c-1] + g[r+1][c]+ g[r+1][c+1]
```

But what do you do if r == 0? There is no -1 row.

The code must deal with the various special cases when the point is on the edge and there is either no adjacent row or no adjacent column. Obviously, some moderately complex conditional tests must be applied to eliminate these cases.

Anything that is "moderately complex" should be promoted into a separate function! An approach that can be used here is to have a `TemperatureAt()` function that is given a row, column position. If the position is within the grid, it returns that temperature. Otherwise it returns "ambient temperature". The "average of neighbors" function can use this `TemperatureAt()` auxiliary function rather than accessing the grid directly.

```
TemperatureAt(row, col)
    if row outside grid
    return ambient
    if col outside grid
           return ambient
    return grid[r][c]

Average of neighbors(row, col)
    ave = - grid[row][col]
    for r=row-1 to row+1
           for col = col-1 to col+1
                  ave += TemperatureAt(r,c)
    return ave/8
```

The sketch for the revised "Average of Neighbors" uses a double loop to run though nine squares on the grid. Obviously that includes the temperature at the centre point when just want to consider its neighbors; but this value can be subtracted away.

Final stages

All that remains is to decide on function prototypes, shared data, and so forth. As in earlier examples, the function prototypes are simplified if we use a typedef to define "Grid" to mean an array of doubles:

```
typedef double Grid[kWIDTH][kLENGTH];
```

The prototypes are:

```
char CodeTemperature(double temp);

void Show(const Grid g);

void CopyGrid(Grid dest, const Grid src);
```

```
double TemperatureAt(const Grid g, int row, int col);

double AverageOfNeighbors(const Grid g, int row, int col);

double NewTemperature(double currenttemp, double envtemp);

void HeatDiffuse(Grid g);

void Initialize(Grid g);

int main();
```

The main `Grid` variable could have been made filescope and shareable by all the functions but it seemed more appropriate to define it as local to main and pass it to the other functions. The const qualifier is used on the `Grid` argument to distinguish those functions that treat it as read only from those that modify it.

There should be no filescope variables, just a few constants. This version of the program uses stream output to display the results so #includes <iostream.h>.

Implementation

The file starts with the #includes and constants. The array `TemperatureSymbols[]` has increasingly 'dense' characters to represent increasingly high temperatures.

```
#include <iostream.h>
#include <assert.h>

const int kWIDTH = 15;
const int kLENGTH = 51;

const double kAMBIENT = 25;
const double kFLAME = 1000;
const double kRANGE = kFLAME - kAMBIENT;
const double kSTEP = kRANGE / 10.0;

const double kCONDUCT = 0.33;

const int kMIDX = kLENGTH / 2;
const int kMIDY = kWIDTH / 2;

const char TemperatureSymbols[] = {
    '.', '-', '+', '=', '*', 'H', '&', '$', '@', '#'
};

typedef double Grid[kWIDTH][kLENGTH];
```

Function `CodeTemperature()` sets the initial range to (25, 120) and keeps incrementing until the range includes the specified temperature. This should result in an index in range 0 .. 9 which can be used to access the `TemperatureSymbols[]` array.

```
char CodeTemperature(double temp)
{
    int i = 0;
    double t = kAMBIENT + kSTEP;
    while(t < temp) { i++; t += kSTEP; }

    assert(i < (sizeof(TemperatureSymbols)/sizeof(char)));

    return TemperatureSymbols[i];
}
```

Both `Show()` and `CopyGrid()` use similar double loops to work through all elements of the array, in one case printing characters and in the other copying values. Show() outputs a '\f' or "form feed" character. This should clear the window (screen) and cause subsequent output to appear at the top of the window. This is more convenient for this form of display than normal scrolling.

The curses library could be used. The routine would then use `CG_PutCharacter( char, int, int)` to output characters at selected positions.

```
void Show(const Grid g)
{
    cout << '\f';
    for(int r=0; r< kWIDTH; r++) {
            for(int c = 0; c < kLENGTH; c++)
                    cout << CodeTemperature(g[r][c]);
            cout << endl;
            }
    cout << endl;
}

void CopyGrid(Grid dest, const Grid src)
{
    for(int r=0;r<kWIDTH;r++)
            for(int c=0;c<kLENGTH; c++) dest[r][c] = src[r][c];
}
```

The `TemperatureAt()` and `AverageOfNeighbors()` functions are straightforward codings of the approaches described in the design:

```
double TemperatureAt(const Grid g, int row, int col)
{
    if(row < 0)
            return kAMBIENT;
    if(row >= kWIDTH)
            return kAMBIENT;
```

```
    if(col < 0)
           return kAMBIENT;

    if(col >= kLENGTH)
           return kAMBIENT;

    return g[row][col];
}

double AverageOfNeighbors(const Grid g, int row, int col)
{
    double Average = -g[row][col];

    for(int r = row - 1; r <= row + 1; r++)
           for(int c = col - 1; c <= col + 1; c++)
                  Average += TemperatureAt(g, r, c);

    return Average / 8.0;
}
```

As explained previously, there is nothing wrong with one line functions if they help clarify what is going on in a program. So function `NewTemperature()` is justified (you could define it as `inline double NewTemperature()` if you didn't like paying the cost of a function call):

```
double NewTemperature(double currenttemp, double envtemp)
{
    return (kCONDUCT*envtemp + (1.0 - kCONDUCT)*currenttemp);
}
```

Function `HeatDiffuse()` has to have a local Grid that holds the current values while we update the main Grid. This copy is passed to `CopyGrid()` to be filled in with current values. Then get the double loop where fill in the entries of the main Grid with new computed values. Finally, reset the centre point to flame temperature.

```
void HeatDiffuse(Grid g)
{
    Grid temp;
    CopyGrid(temp, g);
    for(int r = 0; r < kWIDTH; r++)
           for(int c = 0; c < kLENGTH; c++) {

                  double environment = AverageOfNeighbors(
                                              temp, r, c);
                  double current = temp[r][c];
                  g[r][c] = NewTemperature(
                                       current, environment);
                  }
    g[kMIDY][kMIDX] = kFLAME;
```

```
}

void Initialize(Grid g)
{
    for(int r=0;r<kWIDTH;r++)
           for(int c=0;c<kLENGTH; c++) g[r][c] = kAMBIENT;
    g[kMIDY][kMIDX] = kFLAME;
}
```

The main program is again simple, with all the complex processing delegated to other functions.

```
int main()
{
    int nsteps;
    int nprint;
    cout << "Number of time steps to be modelled : ";
    cin >> nsteps;
    cout << "Number of steps between printouts : ";
    cin >> nprint;

    Grid   g;
    Initialize(g);

    for(int i = 0,  j = 0; i < nsteps; i++) {
           HeatDiffuse(g);
           j++;
           if(j==nprint) { j = 0; Show(g); }
           }
    return 0;
}
```

If you have a very fast computer, you may need to put a delay after the call to `Show()`. However, there are enough floating point calculations being done in the loops to slow down the average personal computer and you should get sufficient time to view one display before the next is generated. A suitable input data values are `nsteps == 100` and `nprint == 10`. The patterns get more interesting if the flame temperature is increased to 5000°C.

12.3 MENU SELECTION

Design of a single utility routine

Many programs need to present their users with a menu of choices. This kind of commonly required functionality can be separated out into a utility routine:

```
int MenuSelect(…);
```

Normal requirements include printing a prompt, possibly printing the list of choices, asking for an integer that represents the choice, validating the choice, printing error message, handling a ? request to get the options relisted, and so forth. This function can be packaged in a little separately compiled file that can be linked to code that requires menu selection facilities.

The function has to be given:

1 an initial prompt ("Choose option for …");

2 an indication as to whether the list of choices should be printed before waiting for input;

3 an array with the strings defining the choices;

4 an integer specifying the number of choices.

The code will be along the following lines:

```
print prompt
if need to list options
    list them
repeat
    ask for integer in specified range
    deal with error inputs
    deal with out of range
until valid entry
```

Dealing with out of range values will be simple, just requires an error message reminding the user of the option range. Dealing with errors is more involved.

Design, second iteration

Errors like end of file or "unrecoverable input" error are probably best dealt with by returning a failure indicator to the calling program. We could simply terminate the program; but that isn't a good choice for a utility routine. It might happen that the calling program could have a "sensible default processing" option that it could use if there is no valid input from the user. Decisions about terminating the program are best left to the caller.

There will also be recoverable input errors. These will result from a user typing alphabetic or punctuation characters when a digit was expected. These extraneous characters should be discarded. If the first character is a '?', then the options should be displayed.

Sorting out unrecoverable and recoverable errors is too elaborate to be expanded in line, we need an auxiliary function. These choices lead to a refined design:

```
handle error
    check input status,
    if end of file or bad then
           return "give up" indicator (?0)
```

```
        read next character
        if ? then
               list options
        remove all input until newline
        return "retry" indicator (?1)

    menuselect
        print prompt
        if need to list options
               list them
        repeat
               ask for integer in specified range
               read input

               if not input good
                      if handle_eror != retry
                             return failure
                      continue

               check input against limits
               if out of range
                      print details of range, and
                             "? to list options"
        until valid entry
        return choice
```

Design, third iteration

The activity "list options" turns up twice, so even though this will probably be a one or two line routine it is worth abstracting out:

```
list options
    for each option in array
        print index number, print option text
```

The `MenuSelect()` function has to be given the prompt and the options. These will be arrays of characters. Again it will be best if a typedef is used to introduce a name that will allow simplification of argument lists and so forth:

```
const int UT_TXTLENGTH = 60;

typedef char UT_Text[UT_TXTLENGTH];
```

The `UT_` prefix used in these names identifies these as belonging to a "utility" group of functions.

Conventions have to be defined for the values returned from the `MenuSelect()` function. A value -1 can indicate failure; the calling routine should terminate the program or find some way of managing without a user selection. Otherwise, the value can be in the range 0 … N-1 if there are N entries in the options table. When

communicating with the user, the options should be listed as 1 … N; but a zero based representation is likely to be more convenient for the calling routine.

Hiding implementation only functions

Although there appear to be three functions, only one is of importance to other programmers who might want to use this menu package. The "list options" and "handle error" routines are details of the implementation and should be hidden. This can be done in C and C++. These functions can be defined as having filescope – their names are not known to any other part of the program.

Finalising the design

The function prototypes can now be defined:

```
static void ListOptions(const UT_Text options[], int numopts);

static int HandleError(const UT_Text options[], int numopts);

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts);
```

The two static functions are completely private to the implementation. Other programmers never see these functions, so their names don't have to identify them as belonging to this package of utility functions. Function `MenuSelect()` and the `Text` (character array) data type are seen by other programmers so their finalised names make clear that they belong to this package.

Implementation

Two files are needed. The "header" file describes the functions and types to other parts of the program (and to other programmers). The implementation file contains the function definitions.

The header file has the usual structure with the conditional compilation bracketing to protect against being multiply included:

UT.h "header file"

```
#ifndef __UT__
#define __UT__

const int UT_TXTLENGTH = 60;

typedef char UT_Text[UT_TXTLENGTH];

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts);

#endif
```

The implementation file will start with #includes on necessary system files and on UT.h. The functions need the stream i/o facilities and the iomanip formatting extras:

UT.cp implementation

```
#include <iostream.h>
#include <iomanip.h>

#include "UT.h"
```

Function `ListOptions()` simply loops through the option set, printing them with numbering (starting at 1 as required for the benefit of users).

```
static void ListOptions(const UT_Text options[], int numopts)
{
    cout << "Choose from the following options:" << endl;
    for(int i = 1; i <= numopts; i++) {
            cout << setw(4) << i << ": " << options[i-1]
                    << endl;
    }
}
```

Function `HandleError()` is only called if something failed on input. It checks for unrecoverable errors like bad input; returning a 0 "give up" result. Its next step is to "clear" the failure flag. If this is not done, none of the later operations would work.

Once the failure flag has been cleared, characters can be read from the input. These will be the buffered characters containing whatever was typed by the user (it can't start with a digit). If the first character is '?', then the options must be listed.

The `cin.get(ch)` input style is used instead of `cin >> ch` because we need to pick up the "whitespace" character '\n' that marks the end of the line. We could only use `cin >> ch` if we changed the "skip whitespace mode" (as explained in Chapter 9); but the mode would have to be reset afterwards. So, here it is easier to use the `get()` function.

```
static int HandleError(const UT_Text options[], int numopts)
{
    if(cin.eof() || cin.bad())
            return 0;

    cin.clear();
    char ch;
    cin.get(ch);
    if(ch == '?')
            ListOptions(options, numopts);
    else
            cout << "Illegal input, discarded" << endl;

    while(ch != '\n')
            cin.get(ch);
```

```
    return 1;
}
```

Function `MenuSelect()` prints the prompt and possibly gets the options listed. It then uses a `do … while` loop construct to get data. This type of loop is appropriate here because we must get an input, so a loop that must be traversed at least once is appropriate. If we do get bad input, `choice` will be zero at the end of loop body; but zero is not a valid option so that doesn't cause problems.

The value of `choice` is converted back to a 0 … N-1 range in the `return` statement.

```
int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
    int numopts, int listopts)
{
    cout <<       prompt << endl;
    if(listopts)
           ListOptions(options, numopts);

    int choice = -1;
    do {
           cout << "Enter option number in range 1 to "
                  << numopts
                  << ", or ? for help" << endl;
           cin >> choice;

           if(!cin.good()) {
                  if(!HandleError(options, numopts))
                         return -1;
                  else continue;
           }

           if(! ((choice >= 1) && (choice <= numopts))) {
                  cout << choice <<
                         " is not a valid option number,"
                         " type ? to get option list" << endl;
           }
    }
    while (! ((choice >= 1) && (choice <= numopts)));
    return choice - 1;

}
```

A simple test program

Naturally, the code has to be checked out. A test program has to be constructed and linked with the compiled UT.o code. The test program will #include the UT.h header file and will define an array of `UT_Text` data elements that are initialized to the possible menu choices:

```
#include <stdlib.h>
#include <iostream.h>
#include "UT.h"

static UT_Text choices[] = {
    "Sex",
    "Drugs",
    "Rock and Roll"
};

int main()
{
    int choice = UT_MenuSelect("Lifestyle?", choices, 3, 0);

    switch(choice) {
case 0:
            cout << "Yes, our most popular line" << endl;
            break;
case 1:
            cout << "Not the best choice" << endl;
            break;
case 2:
            cout << "Are you sure you wouldn't prefer "
                           "the first option?" << endl;
            break;
    }

    return EXIT_SUCCESS;
}
```

When the code has been tested, the UT files can be put aside for use in other programs that need menus.

12.4 PICK THE KEYWORD

Although not quite as common as menu selection, many programs require users to enter keyword commands. The program will have an array of keywords for which it has associated action routines. The user enters a word, the program searches the table to find the word and then uses the index to select the action. If you haven't encountered this style of input elsewhere, you will probably have met it in one of the text oriented adventure games where you enter commands like "Go North" or "Pick up the box".

These systems generally allow the user to abbreviate commands. If the user simply types a couple of letters then this is acceptable provided that these form a the start of a unique keyword. If the user entry does not uniquely identify a keyword, the program can either reject the input and reprompt or, better, can list the choices that start with the string entered by the user.

The function `PickKeyWord()` will need to be given:

1 an initial prompt;

2 an array with the keywords;

3 an integer specifying the number of keyword choices.

The code will be along the following lines:

```
print prompt
loop
    read input
    search array for an exactly matched keyword
    if find match
           return index of keyword
    search array for partial matches
    if none,
           (maybe) warn user that there is no match
           start loop again!
    if single partial match,
           (maybe) identify matched keyword to user
           return index of keyword
    list all partially matching keywords
```

Design, second iteration

The "search array for …" steps are obvious candidates for being promoted into separate auxiliary functions. The search for an exact match can use the `strcmp()` function from the string library to compare the data entered with each of the possible keywords. A partial match can be found using `strncmp()`, the version of the function that only checks a specified number of characters; `strncmp()` can be called asking it to compare just the number of characters in the partial word entered by the user.

Partial matching gets used twice. First, a search is made to find the number of keywords that start with the characters entered by the user. A second search may then get made to print these partial matches. The code could be organized so that a single function could fulfil both roles (an argument would specify whether the partially matched keywords were to be printed). However, it is slightly clearer to have two functions.

The searches through the array of keywords will have to be "linear". The search will start at element 0 and proceed through to element N-1.

Thus, we get to the second more detailed design:

```
FindExactMatch
    for each keyword in array do
           if strcmp() matches keyword & input
                  return index
    return -1 failure indicator

CountPartialMatches
    count = 0
```

```
    for each keyword in array do
           if strncmp() matches input & start of keyword
                  increment count
    return count

PrintPartialMatches
    for each keyword in array do
           if strncmp() matches input & start of keyword
                  print keyword

PickKeyWord
    print prompt
    loop
           read input

           mm = FindExactMatch(…)
           if(mm>=0)
                  return mm

           partial_count = CountPartialMatches(…)

           if partial_count == 0,
                  (maybe) warn user that there is no match
                  start loop again!

           if partial_count == 1,
                  (maybe) identify matched keyword to user
                  return index of keyword ????

           PrintPartialMatches(…)
```

The sketch outline for the main `PickKeyWord()` routine reveals a problem with the existing `CountPartialMatches()` function. We really need more than just a count. We need a count and an index. Maybe `CountPartialMatches()` could take an extra output argument (`int&` – integer reference) that would be set to contain the index of the (last) of the matching keywords.

Design, third iteration

We have to decide what "keywords" are, and what the prompt argument should be. The prompt could be made a `UT_Text`, the same "array of ≈60 characters" used in the MenuSelect() example. The keywords could also be `UT_Texts` but given that most words are less than 12 characters long, an allocation of 60 is overly generous. Maybe a new character array type should be used

```
const int UT_WRDLENGTH = 15;

typedef char UT_WORD[UT_WRDLENGTH ];
```

The `PickKeyWord()` function might as well become another "utility" function and so can share the `UT_` prefix.

As in the case of MenuSelect(), although there four functions, only one is of importance to other programmers who might want to use this keyword matching package. The auxiliary `FindExactMatch()` and related functions are details of the implementation and should be hidden. As before, these functions can be defined as having filescope – their names are not known to any other part of the program.

Finalising the design

The function prototypes can now be defined:

```
static int FindExactMatch(const UT_Word keyws[], int nkeys,
                  const UT_Word input);

static int CountPartialMatches(const UT_Word keyws[], int
nkeys,
                  const UT_Word input, int& lastmatch);

static void PrintPartialMatches(const UT_Word keyws[],
                  int nkeys, const UT_Word input);

int UT_PickKeyWord(const UT_Text prompt,
                  const UT_Word keywords[], int nkeys);
```

These functions can go in the same UT.cp file as the `MenuSelect()` group of functions.

Implementation

The header file UT.h has now to include additional declarations:

```
const int UT_TXTLENGTH = 60;
const int UT_WRDLENGTH = 15;

typedef char UT_Word[UT_WRDLENGTH ];
typedef char UT_Text[UT_TXTLENGTH];

int UT_MenuSelect(const UT_Text prompt, const UT_Text
options[],
           int numopts, int listopts);
int UT_PickKeyWord(const UT_Text prompt,
           const UT_Word keywords[], int nkeys);
```

The implementation in UT.cp is straightforward. Function `FindExactMatch()` has a simple loop with the call to `strcmp()`. Function `strcmp()` returns 0 if the two strings that it is given are equal.

```
static int FindExactMatch(const UT_Word keyws[], int nkeys,
    const UT_Word input)
{
    for(int i = 0; i < nkeys; i++)
```

```
            if(0 == strcmp(input, keyws[i]))
                    return i;
    return -1;
}
```

Functions `CountPartialMatches()` and `PrintPartialMatches()` both use `strlen()` to get the number of characters in the user input and `strncmp()` to compare substrings of the specified length. The count routine has its reference argument `lastmatch` that it can use to return the index of the matched string.

```
int CountPartialMatches(const UT_Word keyws[], int nkeys,
    const UT_Word input, int& lastmatch)
{
    int count = 0;
    int len = strlen(input);
    lastmatch = -1;
    for(int i = 0; i < nkeys; i++)
            if(0 == strncmp(input, keyws[i], len)) {
                    count++;
                    lastmatch = i;
                    }
    return count;
}

void PrintPartialMatches(const UT_Word keyws[], int nkeys,
    const UT_Word input)
{
    cout << "Possible matching keywords are:" << endl;
    int len = strlen(input);
    for(int i = 0; i < nkeys; i++)
            if(0 == strncmp(input, keyws[i], len))
                    cout << keyws[i] << endl;
}
```

The only point of note in the PickKeyWord() function is the "forever" loop – `for(;;) { … }`. We don't want the function to return until the user has entered a valid keyword.

```
int UT_PickKeyWord(const UT_Text prompt,
    const UT_Word keywords[], int nkeys)
{
    cout << prompt;
    for(;;) {
            UT_Word input;
            cin >> input;

            int mm = FindExactMatch(keywords, nkeys, input);
            if(mm>=0)
                    return mm;
```

```
            int match;
            int partial_count =
                   CountPartialMatches(keywords,
                          nkeys, input, match);

            if(partial_count == 0) {
                   cout << "There are no keywords like "
                          << input << endl;
                   continue;
                   }

            if(partial_count == 1) {
                   cout << "i.e. " << keywords[match] << endl;
                   return match;
                   }

            PrintPartialMatches(keywords, nkeys, input);
            }
}
```

A simple test program

As usual, we need a simple test program to exercise the functions just coded:

```
#include <stdlib.h>
#include <iostream.h>
#include "UT.h"

static UT_Word commands[] = {
    "Quit",
    "North",
    "South",
    "East",
    "West",
    "NW",
    "NE",
    "SE",
    "SW"
};

const int numkeys = sizeof(commands)/sizeof(UT_Word);

int main()
{
    cout << "You have dropped into a maze" << endl;

    int quit = 0;

    while(!quit) {
           cout << "From this room, passages lead "
```

```
                               "in all directions" << endl;
            int choice =  UT_PickKeyWord(">",commands, numkeys);
            switch(choice) {
case 1:
case 3:             cout << "You struggle along a steep and"
                                  "narrow passage" << endl;
                    break;
case 2:
case 5:
                    cout << "You move quickly through a rocky"
                                  " passage" << endl;
                    break;
default:
                    cout << "You walk down a slippery, dark,"
                                  "damp, passage" << endl;
                    break;
case 0:
                    quit = 1;
                    break;
                    }

            }
    cout << "Chicken, you haven't explored the full maze"
                              << endl;
    return EXIT_SUCCESS;
}
```

12.5 HANGMAN

You must know the rules of this one. One player selects a word and indicates the number of letters, the other player guesses letters. When the second player guesses a letter that is in the word, the first player indicates all occurrences of the letter. If the second player guesses wrongly, the first player adds a little more to a cartoon of a corpse hung from a gibbet. The second player wins if all letters in the word are guessed. The first player wins if the cartoon is completed while some letters are still not matched.

In this version, the program takes the role of player one. The program contains a large word list from which it selects a word for the current round. It generates a display showing the number of letters, then loops processing letters entered by the user until either the word is guessed or the cartoon is complete. This version of the program is to use the curses functions, and to produce a display like that shown in Figure 12.6.

Specification

Implement the hangman program; use the curses package for display and interaction.

```
+----------------------------------------------------------------------------+
|                                                                            |
|                                                                            |
|    gavotte                                                                 |
|                                              ========                      |
|                                               #   /  |                     |
|                                               #  /   |                     |
|                                               #/     |@                    |
|                                               #    --H---                  |
|                                               #   -  H                     |
|                                               #      H                     |
|                                               #      H                     |
|                                               #     | |                    |
|                                               #    |   |                   |
|                                               #    |   |                   |
|                                               #                            |
|                                               #                            |
|                                             #####                          |
|                                                                            |
+----------------------------------------------------------------------------+

You lost?
```

Figure 12.6 The curses based implementation of the Hangman game

Design

Initial design

This program involves some higher level design decisions than earlier examples where we could start by thinking how the code might work. Here, we have to make some other decisions first; decisions about how to organize the collection of words used by the program, and how to organize the program into files. (This is partly a consequence of the looser specification "implement hangman"; but, generally, as programs get more elaborate you do tend to have to consider more general issues in the design process.)

The words used by the program might be held in a data file. The program would open and read this file at each round, picking "at random" one of the words in the file. However, that scheme has a couple of disadvantage. The program and vocabulary file have to be kept together. When you think how programs get shuffled around different disks, you will see that depending on a data file in the same directory isn't wise. Also, the scheme makes it rather difficult to "pick at random"; until you've read the file you don't know how many choices you've got. So, you can't pick a random number and read that many words from the file and then take the next as the word to use in a round. The words would have to be read into an array, and then one could be picked randomly from the array.

If the words are all going to have to be in an array anyway, we might as well have them as an initialized array that gets "compiled" and linked in with the code. This way avoids problems about the word file getting lost. It seems best to keep this separate from the code, it is easier to edit small files and a long word list would get in the way. So we could have one file "vocab.cp" that contains little more than an initialized array with some common English words, and a compiler set count defining the number of entries in the array.

What are "words"? They small character arrays; like the UT_Word typedef in the last example. In fact we might as well use the UT.h header file with the definition.

Thus the program is going to be built of a number of parts:

- Hangm.cp
  This file will contain the main() function and all the other functions defined for the program.

- vocab.cp
  The file with the words.

- UT.h
  The header file with the typedef defining a "word" as a character array.

- CG.cp and CG.h
  These files contain the curses cursor graphics package.

First iteration through design of code

The program will be something like:

```
initialize random number generator and cursor graphics
loop
    pick a word
    show word as "****" giving indication of length
    loop
           get user to guess character
           check word, where character matches; change
                  displayed word to show matches e.g. "**a*"
           if didn't get any match
                  draw one more part of cartoon
           check for loop termination

    report final win/loss status

    ask if another game,

tidy up
```

Given the relative complexity of the code, this is going to require breaking down into many separate functions. Further, the implementation should be phased with parts of the code made to work while before other parts get implemented.

The main line shown above is too complex. It should be simplified by "abstracting out" all details of the inner loop that plays the game. This reduces it to:

```
main()
    Initialize()
    do
           PlayGame()
    while  AnotherGame()
    TidyUp()
```

This structure should be elaborated with a "reduced" version of `PlayGame()`. The reduced version would just pick a random word, show it a "***", wait for a few second, and then show the actual word. This reduced version allows the basic framework of the program to be completed and tested.

Following these decisions, the preliminary design for the code becomes:

Second iteration, design of a simplified program

```
AnotherGame()
    prompt user for a yes/no reply
    return true if user enters 'y'

PlayGame()
    get curses window displayed
    pick random word
    display word as "****" at some point in window

    !!!delay
    !!!display actual word
```

The two steps marked as !!! are only for this partial implementation and will be replaced later.

Function `AnotherGame()` can be coded using the CG functions like `CG_Prompt()` and `CG_GetChar()`, so it doesn't require further decomposition into simpler functions. However, a number of additional functions will be required to implement even the partial `PlayGame()`.

These functions obviously include a "display word" function and a "pick random word" function. There is also the issue of how the program should record the details of the word to be guessed and the current state of the users guess.

One approach would be to use two "words" – `gGuess` and `gWord`, and a couple of integer counters. The "pick random word" function could select a word from the vocabulary, record its length in one of the integer counters, copy the chose word into `gWord` and fill `gGuess` with the right number of '*'s. The second integer counter would record the number of characters matched. This could be used later when implementing code to check whether the word has been guessed completely.

So, we seem to have:

```
PickRandomWord
    pick random number i in range 0 ... N-1
                  where N is number of word in vocab
    record length of vocab[i] in length
    copy vocab[i] into gWord
    fill gGuess with '*'s

ShowGuess
    move cursor to suitable point on screen
    copy characters from gGuess to screen
```

```
ShowWord
    move cursor to suitable point on screen
    copy characters from gWord to screen
```

Final design iteration for simplified program

The functions identified at this point are sufficiently simple that coding should be straightforward. So, the design of this part can be finished off by resolving outstanding issues of data organization and deciding on function prototypes.

The data consist of the four variables (two words and two integer counters) that can be made globals. If these data are global, then the prototypes for the functions identified so far are:

```
int AnotherGame(void);
void Initialize(void);
void PickWord(void);
void PlayGame(void);
void ShowGuess(void);
void ShowWord(void);

int main()
```

External declarations

The code in the main file Hangm.cp needs to reference the vocabulary array and word count that are defined in the separate vocab.cp file. This is achieved by having "external declarations" in Hangm.cp. An external declaration specifies the type and name of a variable; it is basically a statement to the compiler "*This variable is defined in some other file. Generate code using it. Leave it to the linking loader to find the variables and fill in the correct addresses in the generated code.*"

Implementation of simplified program

The project has include the files Hangm.cp (main program etc), vocab.cp (array with words), and CG.cp (the curses functions); the header files CG.h and UT.h are also needed in the same directory (UT.h is only being used for the definition of the `UT_Word` type).

The file Hangm.cp will start with the #includes of the standard header files and definitions of global variables. The program needs to use random numbers; stdlib provides the `rand()` and `srand()` functions. As explained in 10.10, a sensible way of "seeding" the random number generator is to use a value from the system's clock. The clock functions vary between IDE's. On Symantec, the easiest function to use is `TickCount()` whose prototype is in events.h; in the Borland system, either the function `time()` or `clock()` might be used, their prototypes are in time.h. The header ctype is #included although it isn't required in the simplified program.

```
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
// change events.h to time.h for Borland
```

```
#include <events.h>
#include "CG.h"
#include "UT.h"
```

Here are the extern declarations naming the variables defined in the separate vocab.cp file:

```
extern  UT_Word vocab[];
extern  int numwords;
```

and these are the global variables:

```
UT_Word gGuess;
UT_Word gWord;
int gLength;
int gMatched;
```

The functions `Initialize()`, `AnotherGame()`, `ShowGuess()` and `ShowWord()` are all simple:

```
void Initialize(void)
{
    CG_Initialize();
    // change to srand(time(NULL)); on Borland
    srand(TickCount());
}

int AnotherGame(void)
{
    CG_Prompt("Another Game?");
    char ch = CG_GetChar();
    CG_Prompt("              ");
    return ((ch == 'y') || (ch == 'Y'));
}

void ShowGuess(void)
{
    const xGuess = 6;
    const yGuess = 4;

    CG_MoveCursor(xGuess, yGuess);
    for(int i=0;i<gLength; i++)
           CG_PutCharacter(gGuess[i]);
}

void ShowWord(void)
{
    const xGuess = 6;
    const yGuess = 4;
```

```
    CG_MoveCursor(xGuess, yGuess);
    for(int i=0;i<gLength; i++)
            CG_PutCharacter(gWord[i]);
}
```

Function `PickWord()` uses the random number generator to pick a number that is converted to an appropriate range by taking its value modulo the number of possible words. The chosen word is then copied and the guess word is filled with '*'s as required.

```
void PickWord(void)
{
    int choice = rand() % numwords;
    gLength = strlen(vocab[choice]);
    for(int i = 0; i < gLength; i++) {
            gWord[i] = vocab[choice][i];
            gGuess[i] = '*';
    }
    gWord[gLength] = gGuess[gLength] = '\0';
    gMatched = 0;
}
```

Function `PlayGame()` is supposed to look after a complete game. Of course in this simplified version, it just refreshes the curses window, picks a word, shows it as stars, waits, and shows the word:

```
void PlayGame(void)
{
    CG_FrameWindow();
    PickWord();
    ShowGuess();

    CG_Delay(2);

    ShowWord();

    return;
}
```

Function main() is complete, it shouldn't need to be changed later. The "Tidy()" routine postulated in the original design has collapsed into the call to reset the curses system, there didn't seem to be any other tidying to do so no need for a separate function.

```
int main()
{
    Initialize();
    do {
```

```
            PlayGame();
        }
        while( AnotherGame());

        CG_Reset();
        return 0;
}
```

The other source file is vocab.cp. This should have definitions of the words and the number of words. This file needs to #include UT.h to get the declaration of type UT_Word. The array `vocab[]` should contain a reasonable number of common words from a standard dictionary such as the Oxford English Dictionary:

```
#include "UT.h"

UT_Word vocab[] = {
"vireo",
"inoculum",
"ossuary",
"…
"thurible",
"jellaba",
"whimbrel",
"gavotte",
"clearcole",
"theandric"
};

int numwords = sizeof(vocab)/ sizeof(UT_Word);
```

Although the vocabulary and number of words are effectively constant, they should not be defined as const. In C++, const carries the implication of filescope and the linking loader might not be able to match these names with the external declarations in the Hangm.cp file.

The code as shown should be executable. It allows testing of the basic structure and verifies that words are being picked randomly.

Designing the code to handle the user's guesses

The next step of a phased implementation would be to handle the user's guesses (without exacting any penalties for erroneous guesses).

The `PlayGame()` function needs to be expanded to contain a loop in which the user enters a character, the character is checked to determine whether any additional letters have been matched, and counts and displays are updated.

This loop should terminate when all letters have been guessed. After the loop finishes a "You won" message can be displayed.

The '*'s in the gGuess word can be changed to appropriate characters as they are matched. This will make it easy to display a partially matched word as the `ShowGuess()` function can be called after each change is complete.

Some additional functions are needed. Function `GetGuessedCharacter()` should prompt the user for a character and get input. If the character is a letter, it should be converted to lower case and returned. Otherwise `GetGuessedCharacter()` should just loop, repeating the prompt. Function `CheckCharacter()` should compare the character with each letter in the `gWord` word; if they match, the character should overwrite the '*' in `gGuess` and a count of matches should be incremented. There will also be a `ShowResult()` function that can display a message and cause a short pause.

```
GetGuessed charater
    loop
           prompt for character
           get input
           if letter
                  convert to lower case and return

CheckCharacter ch
    count = 0;
    for each letter in gWord
           if letter == ch
                  count++
                  set character in gGuess

Show Result
    display message
    delay a couple of seconds

PlayGame()
    CG_FrameWindow()
    PickWord()
    ShowGuess()

    gameover = false

    while not gameover
           Get guessed character
           matched = CheckCharacter
           if(matched > 0)
                  ShowGuess
                  gMatched += matched
           gameover = (gMatched == gLength)

    Show Result "You win"
```

The prototypes for the additional functions are:

```
int CheckChar(char ch);
```

```
char GetGuessedChar();
void ShowResult(const char msg[]);
```

Implementation of code for user input

The code implementing these additional and modified function functions is straightforward. Function `GetGuessedCharacter()` uses a do … while loop to get an alphabetic character. Function `CheckChar()` uses a for loop to check the match of letters.

```
char GetGuessedChar()
{
    CG_Prompt("     ");
    CG_Prompt(">");
    char ch;
    do
            ch = CG_GetChar();
    while (!isalpha(ch));
    ch = tolower(ch);
    return ch;
}

int CheckChar(char ch)
{
    int count = 0;
    for(int i = 0; i < gLength; i++)
            if((gWord[i] == ch) && (gGuess[i] == '*')) {
                            gGuess[i] = ch;
                            count++;
                    }
    return count;
}

void ShowResult(const char msg[])
{
    CG_Prompt(msg);
    CG_Delay(5);
}
```

The while loop in `PlayGame()` terminates when `gameOverMan` is true. Currently, there is only one way that the game will be over – all the letters will have been guessed. But we know that the implementation of the next phase will add other conditions that could mean that the game was over.

```
void PlayGame(void)
{
    CG_FrameWindow();
    PickWord();
```

```
    ShowGuess();

    int count = 0;
    int gameOverMan = 0;

    while(!gameOverMan) {
            char ch = GetGuessedChar();
            int matched = CheckChar(ch);
            if(matched > 0)  {
                    ShowGuess();
                    gMatched += matched;
                 }

            gameOverMan = (gMatched == gLength);
    }

    ShowResult("You won");

    return;
}
```

Again, this code is executable and a slightly larger part of the program can be tested.

Designing the code to handle incorrect guesses

The final phase of the implementation would deal with the programs handling of incorrect guesses. Successive incorrect guesses result in display of successive components of the cartoon of the hung man. The game ends if the cartoon is completed.

The cartoon is made up out of parts: the frame, the horizontal beam, a support, the rope, a head, a body, left and right arms, and left and right legs. Each of these ten parts is to be drawn in turn.

The `PlayGame()` function can keep track of the number of incorrect guesses; the same information identifies the next part to be drawn. Selection of the parts can be left to an auxiliary routine – "show cartoon part".

Each part of the cartoon is made up of a number of squares that must be filled in with a particular character. The simplest approach would appear to be to have a little "fill squares" function that gets passed arrays with x, y coordinates, the number of points and the character. This routine could loop using `CG_PutCharacter()` to display a character at each of the positions defined in the arrays given as arguments.

Details of the individual parts are best looked after by separate routines that have their own local arrays defining coordinate data etc.

Thus, this phase of the development will have to deal with the following functions:

```
Fill squares
    loop
```

```
            move cursor to position of next point defined
                   by x, y argument arrays
            output character

Show Frame
    call Fill squares passing that function arrays
            defining the points that make up the frame

Show Head
    call Fill squares passing that function arrays
            defining the points that make up the head

…

Show Cartoon part  (partnumber)
    switch on partnumber
            call show frame, or show beam, or …
            as appropriate
```

Function `PlayGame()` requires some further extension. If a character didn't match, the next cartoon part should be drawn and the count of errors should be increased. There is an additional loop termination condition – error count exceeding limit. The final display of the result should distinguish wins from losses.

```
PlayGame()
    CG_FrameWindow()
    PickWord()
    ShowGuess()

    gameover = false
    count = 0;
    while not gameover
            Get guessed character
            matched = CheckCharacter
            if(matched > 0)
                   ShowGuess
                   gMatched += matched
            else
                   ShowCartoonPart
                   count++

            gameover = (gMatched == gLength) || (count > limit)

    if (gMatched== gLength)Show Result "You win"
    else
            ShowWord
            Show Result "You lost"
```

Implementation of final part of code

The extra routines that get the cartoon parts drawn are all simple, only a couple of representatives are shown:

```
void FillSquares(char fill, int num, int x[], int y[])
{
    for(int i = 0; i < num; i++) {
           CG_MoveCursor(x[i],y[i]);
           CG_PutCharacter(fill);
    }
}

void ShowBeam(void)
{
    int x[] = { 51, 52, 53, 54, 55, 56, 57, 58 };
    int y[] = {  5,  5,  5,  5,  5,  5,  5,  5 };
    int n = sizeof(x) / sizeof(int);
    FillSquares('=', n, x, y);
}

void ShowHead(void)
{
    int x[] = { 59 };
    int y[] = {  8 };
    int n = sizeof(x) / sizeof(int);
    FillSquares('@', n, x, y);
}


void ShowCartoonPart(int partnum)
{
    switch(partnum) {
case 0:
           ShowFrame();
           break;
case 1:
           ShowBeam();
           break;
case 2:
           ShowSupport();
           break;
…
…
case 9:
           ShowRightLeg();
           break;
    }
}
```

The final version of `PlayGame()` is:

```
void PlayGame(void)
{
    CG_FrameWindow();
    PickWord();
    ShowGuess();

    int count = 0;
    int gameOverMan = 0;

    while(!gameOverMan) {
            char ch = GetGuessedChar();
            int matched = CheckChar(ch);
            if(matched > 0)  {
                    ShowGuess();
                    gMatched += matched;
                }
            else {
                    ShowCartoonPart(count);
                    count++;
                    }
            gameOverMan = (count >= kNMOVES) ||
                                (gMatched == gLength);
    }

    if(gMatched==gLength) ShowResult("You won");
    else {
            ShowWord();
            ShowResult("You lost");
    }
    return;
}
```

(The constant kNMOVES = 10 gets added to the other constants at the start of the file.)

Phased implementation strategy

You will find that most of the larger programs that you must write will need to be implemented in phases as was done in this example.

12.6 LIFE

Have you been told "You are a nerd – spending too much time on your computer."? Have you been told "Go out and get a life."?

OK. Please everyone. Get a Life in your computer – Conway's Life.

Conway's Life is not really a computer game. When it was first described in Scientific American it was introduced as a "computer recreation" (Sci. Am. Oct 1970, Feb 1971). The program models a "system" that evolves in accord with specified rules. The idea of the "recreation" part is that you can set up different initial states for this "system" and watch what happens.

The system consists of a two-dimensional array of cells. This array is, in principle, infinite in its dimensions; of course for a computer program you must chose a finite size. Cells contain either nothing, or a "live organism". Each cell (array element) has eight neighbors – three in the row above, three in the row below, a left neighbor and a right neighbor.

Evolution proceeds one generation at a time. The rules of the recreation specify what happens to the contents of each cell at a change of generation. The standard rules are:

1 An organism in a cell "dies of loneliness" if it has less than two neighbors.

2 An organism "dies of overcrowding" if has four or more neighbors.

3 An organism survives for another generation if it has two or three neighbors.

4 An empty cell is filled with a new live organism if it had exactly three neighbors.

These deaths and births are coordinated. They happen simultaneously (this affects how the simulation may be programmed).

The recreation is started by assigning an initial population of organisms to selected cells. Most distributions are uninteresting. The majority of the population quickly die out leaving little groups of survivors that cling tenaciously to life. But some patterns have interesting behaviours. Figure 12.7 shows three examples – "Cheshire Cat", "Glider", and "Tumbler"; there are others, like "glider guns" and "glider eaters" that are illustrated in the second of the two Scientific American articles. A "Glider" moves slowly down and rightwards; a "Cheshire Cat fades away to leave just a grin".

```
  Cheshire Cat                                  Tumbler
...................................................................
......*..*......................................**.**..............
......****......................................**.**..............
.....*....*......................................*.*...............
.....*.**.*...................................*..*.*..*............
.....*....*...................................*..*.*..*............
......****.....................................**...**.............
...................................................................
......................*.*..........................................
.......................**..........................................
.......................*...........................................
...................................................................
...................................................................
...................................................................
                        Glider
```

Figure 12.7 Configurations for Conway's Life

Specification

Implement a version of Conway's Life recreation. The program should use the curses package for display. It should start with some standard patterns present in the array. The program should model a small number of generational turns, then ask the user whether to continue or quit. The modelling and prompting process continues until the user enters a quit command.

Design

First Iteration

This program will again have multiple source files – Life.cp and CG.cp (plus the CG.h header). All the new code goes in the Life.cp file. The code that scans the array and sorts out which cells are "live" at the next generation will have some points in common with the "heat diffusion" example.

The overall structure of the program will be something like:

```
Initialize curses display and "life arrays"
Place a few standard patterns in cells in the array
display starting configuration
loop
    inner-loop repeated a small number of times (≈5)
           work out state for next generation
           display new state
    ask user whether to continue or quit
tidy up
```

As always, some steps are obvious candidates for being promoted into separate functions. These "obvious" functions include an initialization function, a to set an initial configuration, a display function, a "run" function that looks after the loops, and a function to step forward one generation.

The "life arrays" will be similar to the arrays used in the heat diffusion example. As in that example, there have to be two arrays (at least while executing the function that computes the new state). One array contains the current state data; these data have to be analyzed to allow new data to be stored in the second array. Here the arrays would be character arrays (really, one could reduce them to "bit arrays" but that is a lot harder!). A space character could represent and empty cell, a '*' or '#' could mark a live cell.

This example can illustrate the use of a "three dimensional array"! We need two nxm arrays (current generation, next generation). We can define these as a single data aggregate:

```
const int kROWS = 20;
const int kCOLS = 50;

char gWorlds[2][kROWS][kCOLS];
```

We can use the two `[kROWS][kCOLS]` subarrays alternately. We can start with the initial life generation in subarray `gWorlds[0]`, and fill in `gWorlds[1]` with details for the next generation. Subarray `gWorlds[1]` becomes the current generation that gets displayed. At the next step, subarray `gWorlds[0]` is filled in using the current `gWorlds[1]` data. Then `gWorlds[0]` is the current generation and is displayed. As each step is performed, the roles of the two subarrays switch. All the separate functions would share the global `gWorlds` data aggregate.

Second iteration through design process

Each function has to be considered in more detail, possibly additional auxiliary functions will be identified. The initial set of functions become:

```
Initialize
    fill both gWorld[] subarrays with spaces
           (representing empty cells)
    initialize curses routines

Set starting configuration
    place some standard patterns
           Not specified, so what patterns?
           maybe a glider at one point in array, cheshire cat
                  somewhere else.

Display state
    double loop through "current gWorld" using
           curses character display routines to plot spaces
                  and stars

Run
    Display current state
    loop
           ask user whether to quit,
                  and break loop if quit command entered
           loop ≈ 5 times
                  step one generation forward
                  display new sate

Step
    ?

main()
    Initialize
    Set Starting Configuration
    Run
    reset curses
```

This second pass through the program design revealed a minor problem. The specification requires "some standard patterns present in the array"; but doesn't say what patterns! It is not unusual for specifications to turn out to be incomplete. Usually, the designers have to go back to the people who commissioned the program to get more

details specified. In a simple case, like this, the designers can make the decisions – so here "a few standard patterns" means a glider and something else.

There are still major gaps in the design, so it is still too early to start coding.

The "standard patterns" can be placed in much the same way as the "cartoon components" were added in the Hangman example. A routine, e.g. "set glider", can have a set of x, y points that it sets to add a glider. As more than one glider might be required, this routine should take a pair of x, y values as an origin and set cells relative to the origin.

Some of the patterns are large, and if the origin is poorly chosen it would be easy to try to set points outside the bounds of the array. Just as in the heat diffusion example where we needed a TemperatureAt(row, column) function that checked the row and column values, we will here need a SetPoint(row, column) that sets a cell to live provided that the row and column are valid.

These additions make the "set starting configuration" function into the following group of functions:

```
set point (col, row)
    if col and row valid
           add live cell to gWorld[0][row][col]

set glider (xorigin, yorigin)
    have arrays of x, y values
    for each entry in array
           set point(x+xorigin, y + yorigin)

Set starting configuration
    set glider …, …
    set cc …, …
    …
```

(Note, x corresponds to column, y corresponds to row so if set point is to be called with an x, y pair then it must define the order of arguments as column, row.)

The other main gap in the second design outline was function `Step()`. Its basic structure can be obtained from the problem description. The entire new generation array must be filled, so we will need a double loop (all the rows by all the columns). The number of neighbors of each cell in current `gWorld` must be calculated and used in accord with the rules to determine whether the corresponding cell is 'live' in the next `gWorld`. The rules can actually be simplified. If the neighbor count is not 2 or 3, the cell is empty in the next generation (0, 1 imply loneliness, 4 or more imply overcrowding). A cell with 3 neighbors will be live at next generation (either 3 neighbors and survive rule, or 3 neighbors and birth rule). A cell with 2 neighbors is only live if the current cell is live. Using this reformulated version of the rules, the `Step()` function becomes:

```
Step
    identify which gWorld is to be filled in
```

```
    for each row
           for each col
                  nbrs = Count live Neighbors of cell[row][col]
                         in current gWorld
                  switch(nbrs)
                  2 new cell state = current cell state
                  3 new cell state = live
                  default new cell state = empty
    note gWorld just filled in as "current gWorld"
```

The `Step()` function and the display function both need to know which of the gWorlds is current (i.e. is it the subarray `gWorld[0]` or `gWorld[1]`). This information should be held in another global variable `gCurrent`. The `Step()` function has to use the value of this variable to determine which array to fill in for the next generation, and needs to update the value of `gCurrent` once the new generation has been recorded.

We also need a "count live neighbors" function. This will be a bit like the "Average of neighbors" function in the Heat Diffusion example. It will use a double loop to run through nine elements (a cell and its eight neighbors) and correct for the cell's own state:

```
Count Neighbors (row, col)
    count = -Live(row,col)
    for r = row - 1 to row + 1
           for c = col - 1 to col + 1
                  cout += Live(r,c)
```

Just as the Heat Diffusion example needed a `TemperatureAt(row, col)` function that checked for out of range row and col, here we need a `Live(row, col)` function. What happens at the boundaries in a finite version of Life? One possibility is to treat the non-existent cells (e.g. cells in row == -1) as being always empty; this is the version that will be coded here. An alternative implementation of Life "wraps" the boundaries. If you want a cell in `row == kROWS` (i.e. you have gone off the bottom of the array), you use row 0. If you want a cell in `row == -1`, (i.e. you have gone off the top of the array), you use `row == kROWS-1` (the bottommost row). If you "wrap" the world like this, then gliders moving off the bottom of the array reappear at the top.

The code for functions like `Live()` will be simple. It can use a `Get()` function to access an element in the current `gWorld` (the code of `Get()` determines whether we are "wrapping" the `gWorld` or just treating outside areas as empty). `Live()` will simply check whether the character returned by `Get()` signifies a 'live cell' or an 'empty cell'. So, no further decomposition is required.

Third iteration through design process

The function prototypes and shared data can now be specified. The program needs to have as globals an integer `gCurrent` that identifies which `gWorld` subarray is current, and, as previously mentioned, the `gWorld` data aggregate itself.

The functions are:

```
char Get(int row, int col);
```

```
int CountNbrs(int row, int col);

void DisplayState(void);

void Initialize(void);

int Live(int row, int col);

void Run(void);

void SetPoint(int col, int row);

void SetGlider(int x0, int y0);

void SetStartConfig(void);

void Step(void);

int main();
```

Implementation

For the most part, the implementation code is straightforward. The files starts with the #includes; in this case just ctype.h (need `tolower()` function when checking whether user enters "quit command") and the CG header for the curses functions. After the #includes, we get the global variables (the three dimensional `gWorld` aggregate etc) and the constants.

```
#include <ctype.h>
#include "CG.h"

const int kROWS = 20;
const int kCOLS = 50;
const int kSTEPSPERCYCLE = 5;

char gWorlds[2][kROWS][kCOLS];

int gCurrent = 0;

const char EMPTY = ' ';
const char LIVE = '*';
```

Functions `Initialize()` and `DisplayState()` both use similar double loops to work through all array elements. Note the `c+1, r+1` in the call to `CG_PutCharacter()`; the arrays are zero based but the display system is one based (top left corner is square 1, 1 in display).

```
void Initialize(void)
{
    for(int w=0; w < 2; w++)
            for(int r = 0; r < kROWS; r++)
                    for(int c = 0; c < kCOLS; c++)
                                    gWorlds[w][kROWS][kCOLS] =
                                            EMPTY;

    CG_Initialize();
}

void DisplayState(void)
{
    for(int r = 0; r < kROWS; r++)
            for(int c = 0; c < kCOLS; c++)
                    CG_PutCharacter(gWorlds[gCurrent][r][c],
                                    c+1, r+1 );
}
```

Function `Get()` looks indexes into the current `gWorld` subarray. If "wrapping" was needed, a `row` value less than 0 would have `kROWS` added before accessing the array; similar modifications in the other cases.

Function `Live()` is another very small function; quite justified, its role is different from `Get()` even if it is the only function here that uses `Get()` they shouldn't be combined.

```
char Get(int row, int col)
{
    if(row < 0)
            return EMPTY;
    if(row >= kROWS)
            return EMPTY;
    if(col < 0)
            return EMPTY;
    if(col >= kCOLS)
            return EMPTY;

    return gWorlds[gCurrent][row][col];
}

int Live(int row, int col)
{
    return (LIVE == Get(row,col));
}
```

Function `CountNbrs()` is very similar to the `AverageOfNeighbors()` function shown earlier:

```
int CountNbrs(int row, int col)
```

```
{
    int count = -Live(row,col); // Don't count self!
    for(int r = row - 1; r <= row + 1; r++)
            for(int c = col - 1; c <= col + 1; c++)
                    count += Live(r,c);
    return count;
}
```

Function `Step()` uses a "tricky" way of identifying which `gWorld` subarray should be updated. Logically, if the current `gWorld` is subarray `[0]`, then subarray `gWorld[1]` should be changed; but if the current `gWorld` is `[1]` then it is `gWorld[0]` that gets changed. Check the code and convince yourself that the funny modulo arithmetic achieves this:

```
void Step(void)
{
    int Other = (gCurrent + 1) % 2;
    for(int r = 0; r < kROWS; r++)
            for(int c = 0; c < kCOLS; c++) {
                    int nbrs = CountNbrs(r,c);
                    switch(nbrs) {
case 2:
                            gWorlds[Other][r][c] =
                                    gWorlds[gCurrent][r][c];
                            break;
case 3:
                            gWorlds[Other][r][c] = LIVE;
                            break;
default:
                            gWorlds[Other][r][c] = EMPTY;
                            }
                    }
    gCurrent = Other;
}
```

Function `SetPoint()` makes certain that it only accesses valid array elements. This function might need to be changed if "wrapping" were required.

```
void SetPoint(int col, int row)
{
    if(row < 0)
            return;
    if(row >= kROWS)
            return;
    if(col < 0)
            return;
    if(col >= kCOLS)
            return;

    gWorlds[gCurrent][row][col] = LIVE;
```

```
}
```

Function `SetGlider()` uses the same approach as the functions for drawing cartoon parts in the Hangman example:

```
void SetGlider(int x0, int y0)
{
    int x[] = { 1, 2, 3, 1, 2 };
    int y[] = { 1, 2, 2, 3, 3 };
    int n = sizeof(x) / sizeof(int);
    for(int i = 0; i < n; i++)
            SetPoint(x0 + x[i], y0 + y[i]);
}

void SetStartConfig(void)
{
    SetGlider(4,4);
    …
}
```

Function `Run()` handles the user controlled repetition and the inner loop advancing a few generation steps:

```
void Run(void)
{
    char ch;
    DisplayState();
    int quit = 0;
    for(;;) {
            CG_Prompt("Cycle (C) or Quit (Q)?");
            ch = CG_GetChar();
            ch = tolower(ch);

            if(ch == 'q')
                    break;
            CG_Prompt("                        ");

            for(int i = 0; i < kSTEPSPERCYCLE; i++) {
                    Step();
                    DisplayState();
                    }
            }
}
```

As usual, `main()` should be basically a sequence of function calls:

```
int main()
{
    Initialize();
    SetStartConfig();
```

```
    Run();
    CG_Reset();
    return 0;
}
```

EXERCISES

1 Use the curses package to implement a (fast) clock:

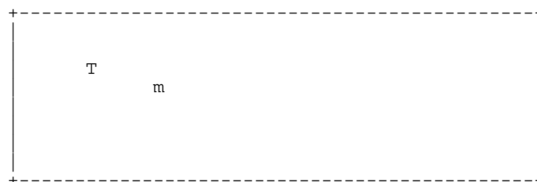
```
.........*****.........
.....*...........*.....
...*........#.......*..
.*..........#........*.
*..........#..........*
*..........#..........*
*...........@.........*
*............@........*
.*............@......*.
...*...........@....*..
.....*..........@*.....
.........*****.........
```

The "minutes" hand should advance every second, and the "hours" hand every minute.

2 In his novel "The Day of the Triffids", John Wyndham invented triffids – hybrid creatures that are part plant, part carnivore. Triffids can't see, but they are sensitive to sound and vibrations. They can't move fast, but they can move. They tend to move toward sources of sounds – sources such as animals. If a triffid gets close enough to an animal, it can use a flexible vine- like appendage to first kill the animal with a fast acting poison, and then draw nutrients from the body of the animal

At one point in the story, there is a blind man in a field with a triffid. The man is lost stumbling around randomly. The triffid moves (at less than one fifth of the speed of the man), but it moves purposively toward the source of vibrations – the triffid is hunting its lunch.

Provide a graphical simulation using the curses library.



3 Somewhere, on one of the old floppies lost in your room is a copy of a 1980s computer game "Doctor Who and the Daleks". Don't bother to search it out. You can now write your own version.

The game consist of a series of rounds with the human player controlling the action of Dr. Who, and the program controlling some number of Daleks. A round is comprised on many turns. The human player and the program take alternate turns; on its turn the program

will move all the Daleks. Things (Daleks and Dr) can move one square vertically, horizontally, or diagonally. Nothing can move out of the playing area. The display is redrawn after each turn. Dr. Who's position is marked by a W, the Daleks are Ds, and wrecked Daleks are #s.

```
+---------------------------------------------+
|                                             |
|     D                 D                     |
|            W  #                             |
|                               D             |
|                                             |
|                                             |
|                                             |
+---------------------------------------------+
Command>
```

The human player can enter commands that move Dr. Who (these are one character commands, common choices of letter and directions are 'q' NW, 'w' N, 'e' NE, 'a' W, … with North being up the screen).

If a Dalek moves onto the square where Dr. Who is located, the game ends. If a Dalek moves onto a square that is occupied, it is wrecked and does not participate further in the round. The wreckage remains for the rest of that round; other Daleks may crash into wreckage and destroy themselves. Dr. Who can not move onto a square with wreckage.

The Daleks' moves are determined algorithmically. Each Dalek will move toward the current position of Dr. Who. The move will be vertical, horizontal, or diagonal; it is chosen to minimize the remaining distance from the Dalek's new position to that of Dr. Who.

A round of the game starts with Dr. Who and some number of Daleks being randomly placed on a clear playing area. (There are no constraints, Dr. Who may discover that he is totally surrounded by Daleks; he may even start dead if a Dalek gets placed on his square.)

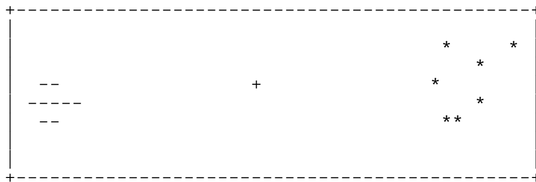
The human player's winning strategy is to manoeuvre Dr. Who so as to cause the maximum number of collisions among Daleks.

The player has two other options. Dr. Who may be teleported to some random location (command 't') so allowing escape from converging Daleks. There are no guarantees of safe arrival; he may rematerialise on wreckage (and be killed), on a Dalek (and again be killed), or next to a Dalek (and be killed when the Dalek moves). Once per round, Dr. Who may use a "sonic screwdriver" that destroys Daleks in the immediate vicinity (immediate? radius 2, radius 3, depends how generous you feel).

If Dr. Who survives a round in which all Daleks are destroyed, a new round starts. Each new round increases the number of Daleks up to a maximum. Initially, there are three Daleks, the maximum depends on the size of the playing area but should be around 20.

4. The professor who commissioned the π-cannon is disappointed at the rate of convergence and believes that the cannoneers are doing something wrong.

   Combine the code from the π-cannon example with the curses package to produce a visual display version that the professor may watch:
   The program should show the cannon ball moving from gun barrel to point of impact (getting the successive x positions is easy, you need to do a bit of trigonometry to get the y positions).



5. Modify Conway's Life so that the playing area is "wrapped" (i.e. gliders moving off the top of the play area reappear at the bottom etc.)

6. Implement a version of the "Mastermind" game using the curses package.

   In "Mastermind", the program picks a four digit (or four letter code), e.g. EAHM. The human player tries to identify the code in the minimum number of guesses. The player enters a guess as four digits or letters. The program indicates 1) how many characters are both correct and in the correct place, and 2) how many characters are correct but in the wrong place (these reports do not identify which characters are the correct ones). The player is to use these clues to help make subsequent guesses.

   A display shows the history of guesses and resulting clues:

```
Guess# Correct       Guess         Misplaced letter
 1                     ABCD               *
 2                     WXYZ
 3         *           EFGH               *
 4        **           EAIJ
```

   Rules vary as to whether you can have repeated characters. Make up your own rules.

7. Build a better curses.

   The "refresh" rate when drawing to a window is poor in the curses package given in the text.

   You can make things faster by arranging that the curses package make use of a character array screen[kHEIGHT][kWIDTH]. This is initialized to contain null characters. When characters are drawn, a check is made against this array. If the array element corresponding to the character position has a value that differs from the required character, then the required character is drawn and stored in the array. However, if the array indicates that the character on the screen is already the required character then no drawing is necessary.

8. Combine the Life program with the initial curses drawing program to produce a system where a user can sketch an initial Life configuration and then watch it evolve.