29

29 The Power of Inheritance and Polymorphism

This chapter is another that presents just a single example program illustrating how to apply features of the C++ language. In this case, the focus is on class inheritance and the use of polymorphism. The application program is a simple game.

In the days before "Doom", "Marathon", and "Dark Forces", people had to be content with more limited games! The basic game ideas were actually rather similar. The player had to "explore a maze", "find items", and avoid being eliminated by the "monsters" that inhabited the maze. However, details of graphics and modes of interaction were more limited than the modern games. A maze would be represented as an array of characters on the screen (walls represented by '#' marks, staircases between levels by characters like '<' and so forth). Single character commands allowed the player to move the maze-explorer around within the boundaries of the maze. "Rogue" and "Moria" are typical examples of such games; you may find that you have copies on old floppies somewhere.

You aren't quite ready to put Lucas Arts out of business by writing a replacement for "Dark Forces", but if you have got this far you can write your own version of Moria.

Example program

The example code given here is simplified with lots of scope for elaboration. It employs a single level maze (or "dungeon"). A map of the dungeon is always displayed, in full, on the screen. The screen size limits the size of the map; having the complete map displayed simplifies play. (Possible elaborations include showing only those parts of the map already explored and "scrolling" maps that are too large to fit on a screen.) The map and other data are displayed using the same system features as used in the "cursor graphics" examples from Chapter 12.

The map, details of the items to be found, and the monsters that are to be avoided (or destroyed) are taken from text file input. Again, this is a simplification. Games like Moria usually generate new maps for every game played.

The playing mechanism is limited. The user is prompted for a command. After a user command has been processed, all the "monsters" get a chance to "run". A "Monster:: Run()" function captures the essence of "monsterdom" i.e. a desire to eliminate the human player.

Objects everywhere

Naturally, such a game program requires numerous obvious objects – the "monsters", the "items" that must be collected, the "player" object, the dungeon object itself. In addition there will have to be various forms of "window" object used to display other information. Since there will be many "monsters" and many "items", standard collection classes will be needed.

Windows hierarchy

There are two separate class hierarchies as well as a number of independent classes. One limited class hierarchy defines "windows". There is a basic "window" class for display of data with a couple of specializations such as a window used to output numeric data (e.g. player's "health") and a window to get input. (These "window" classes are further elaborated in the next chapter.)

"Dungeon Items" hierarchy

There is also a hierarchy of "things found in the dungeon". Some, like the "items" that must be collected, are passive, they sit waiting until an action is performed on them. Others, like the player and the monsters, are active. At each cycle of the game, they perform some action.

Polymorphic pointers

Naturally, there are many different kinds of monster. Each different kind (subclass) employs a slightly different strategy when trying to achieve their common objective of "eliminating the player". This is where the polymorphism comes in. The "dungeon" object will work with a collection of monsters. When it is the monsters' turn, the code has a loop that lets each monster "run" (`Monster *m; …; m->Run();`). The pointer `m` is polymorphic – pointing to different forms of monster at different times. Each different form has its own interpretation of `Run()`.

29.1 THE "DUNGEON" GAME

The "dungeon" game program is to:

- Read game details from a text file. These details are to include the map layout, the initial positions and other data defining collectable items, the monsters, and the player.
- Provide a display similar to that shown in Figure 29.1. This display is to include the main map window (showing fixed features like walls, position of collectable items, and current positions of active items) and other windows that show the player's status.
- Run the game. The game terminates either by the player object acquiring all collectable items or by its "health" falling to zero.
- Operate a "run cycle" where the user enters a movement command (or "magic action" command – see below), the command is executed, and then all other active items get a chance to run.
- Arrange that the player object acquire a collectable item by moving over the point where the item is located. Acquisition of a collectable item will change one or more of the player object's "health", "wealth", or "manna" attributes. Once taken by the player object, collectable items are to be deleted from the game.
- Employ a scheme where single character commands identify directional movements or "magic actions" of the player.

Main window (map display)   "Monsters"   Collectable Items

```
+----------------------------------------------------------------+
|w       #   w    #         #      g #        # $      *     = # |
| ###### #        #  #####  #        ######## # #######   ###### |
| #      # ########  #   #  #   #### #  #     # #   $ # #   # #  |
| # #### # #  w   #  #   #  # p #  # #  #  #  # #       #   # #  |
| # #    # #      #    $ #  #   #  # #  #  # ##########w### # #  |
| # # ## # # ######  #####  #   #  # #  #### #        # #     #  |
|   #  # # # #           #  #  ##  # #       #          # ##### #|
| # #  # # # # ######### #  #  #  ## ############ ##### # #      |
| # #  # # #   #         #  #  #  #  #            #   # # # #### |
| # # ## # # ### ######        #       #########  #     # # #  # |
| # #        # # #    #  ##### # #######       #      # # # # ## |
|   # ######## # #    #  #   # # #     #       #   =  #   # # #  |
|####          # #    #  #   # # # g   #              # # # # #  |
|     ######   #   ####  #       #       #######      # # # # #  |
|     # d  #   #         #################            # # # # #  |
|   $ # $  #   #               #             g  #     # # # #    |
|   $ #        #        h      #                #       # #   ## |
|     # ####   #               #               ########          |
+----------------------------------------------------------------+
 |Health          37|   |Manna           6|     |Wealth          30|
 +-----------------+   +-----------------+     +-----------------+
 |Direction  4     |
 +-----------------+
```

Number window

Edit window   Player object

Figure 29.1 "Dungeon" game's display.

- Handle attacks by monster objects on the player. A monster object adjacent to the player will inflict damage proportional to its strength. This amount of damage is deducted from the player object's health rating. Some monster objects have the option of using a projectile weapon when not immediately adjacent to the player.

- Handle attacks by the player on a monster object. A movement command that would place the player object on a point occupied by a monster is to be interpreted as an attack on that monster. The player inflicts a fixed amount of damage. Such an attack reduces the monster's health attribute. If a monster object's health attribute falls to zero, it is to be deleted from the game.

- Handle movements. The player and most types of monster are limited in their scope for movement and cannot pass through walls or outside of the map area. More than one monster may be located on the same point of the map; monsters are allowed to occupy the same points as collectable items. When several dungeon items are located at the same point, only one is shown on the map.

  The player's health and manna ratings increase slowly as moves are made.

- Support "magic action" commands. Magic action commands weaken or destroy monsters at a distance from the player; like movement commands, they are directional.

  A magic action command inflicts a predefined amount of damage on any monster located on the neighbouring square in the specified direction, half that amount of damage on any monster two squares away along the specified directional axis, a quarter that amount on a monster three squares away etc. Magic does not project through walls.

  Use of a magic action command consumes "manna" points. If the player object has insufficient manna points, the player suffers damage equal to twice the deficit. So, if a command requires 8 manna points and the player object's manna is 3, the manna is reduced to zero and the player object's health is reduced by 10 after executing the command.

- Provide the following basic behaviours for monster objects.

  A monster object will attack the player object if it is on an adjacent point.

  If not immediately adjacent to the player object, some monsters "look" for the player and, if they can "see" the player, they may advance toward it or launch a projectile.

  If they are not able to detect the player object, a monster object will perform its "normal movement" function. This might involve random movement, no movement, or some more purposive behaviour.

  Monster objects do not attempt to acquire collectable items.

  Monster objects do not interact with other monster objects.

29.2 DESIGN

29.2.1 Preliminaries

This "preliminaries" section explores a few aspects of the program that seem pretty much fixed by the specification. The objective is to fill out some details and get a few pointers to things that should be considered more thoroughly.

For example the specification implies the existence of "class Dungeon", "class Player", "class Monster", a class for "collectable items" and so forth. We might as well jot down a few initial ideas about these classes, making a first attempt to answer the perennial questions "*What does class X do? What do instances of class X own?*". Only the most important responsibilities will get identified at this stage; more detailed consideration of specific aspects of the program will result in further responsibilities being added. Detailed analysis later on will also show that some of the classes are interrelated, forming parts of a class hierarchy.

Other issues that should get taken up at this preliminary stage are things like the input files and the form of the main program. Again, they are pretty much defined by the specification, but it is possible to elaborate a little.

main()

We can start with the easy parts – like the `main()` function! This is obviously going to have the basic form "create the principal object, tell it to run":

```
int main()
{
    Dungeon *d;
    d = new Dungeon;

    Prompt user for name of file and read in name
    …

    d->Load(aName);

    int status = d->Run();
    Terminate(status);

    return 0;
}
```

The principal object is the "Dungeon" object itself. This has to load data from a file and run the game. When the game ends, a message of congratulations or commis-erations should be printed. The `Dungeon::Run()` function can return a win/lose flag that can be used to select an appropriate message that is then output by some simple `Terminate()` function.

First idea for files

The files are to be text files, created using some standard editor. They had better specify the size of the map. It would be simplest if the map itself were represented by the '`#`' and ' ' characters that will get displayed. If the map is too large, the top-left portion should be used.

Following the map, the file should contain the data necessary to define the player, the collectable items, and the monsters. The program should check that these data define exactly one player object and at least one collectable item. The program can simply terminate if data are invalid. (It would help if an error message printed before termination could include some details from the line of input where something invalid was found.)

Collectable items and other objects can be represented in the file using a character code to identify type, followed by whatever number of integers are needed to initialize an object of the specified type. A sentinel character, e.g. 'q', can mark the end of the file.

A plausible form for an input file would be:

```
width and height (e.g 70 20)
several (20) lines of (70) characters, e.g.
##################### … … ###########
#          #        # … …        #  #
# ######## #        # … …   ####### #
dungeon items
h 30 18 …          human (i.e. player), coords, other data
w 2 2 10 …         wandering monster, coords, …
…
$ 26 6 0 30 0      collectable item, coords, values
q                  end mark
```

Any additional details can be resolved once the objects have been better characterized.

class Dungeon

Consideration of the `main()` function identified two behaviours required of the `Dungeon` object: loading a file, and running the game.

The `Dungeon::Load()` function will be something along the following lines:

```
Dungeon::Load
    Open file with name given
    Load map
    Load other data

Dungeon:: load map
    read size
    loop reading lines of characters that define
           the rows of the map

Dungeon:: load other data
    read type character
    while character != 'q'
           create object of appropriate type
           tell it to read its own data
           if it is a monster, add to monster collection
           if it is a collectable, add to collectable
                  items collection
           if player, note it (make sure no existing player)

    check that the data defined some items to collect
```

The `Dungeon::Run()` function could have the following general form:

```
Dungeon::Run()
    Finalize setting up of displays
    Draw initial state

    while(player "alive")
           player "run"
```

```
        if(all collectables now taken)
              break;

        for each Monster m in monster collection
              m->Run();

  return (player "alive");
```

The displays must be set up. Obviously, the `Dungeon` object owns some window objects. (Some might belong to the `Player` object; this can be resolved later.) The `Dungeon` object will get primary responsibility for any work needed to set up display structures.

The main loop has two ways of terminating – "death" of player, and all collectable objects taken. The game was won if the player is alive at the end.

The `Dungeon` object owns the collection of monsters, the collection of collectable items, and the player object. Collections could use class `List` or class `DynamicArray`.

The `Player` object will need to access information held by the `Dungeon` object. For example, the `Player` object will need to know whether a particular square is accessible (i.e. not part of a wall), and whether that square is occupied by a collectable item or a monster. When the `Player` takes a collectable item, or kills a monster, the `Dungeon` should be notified so that it can update its records. Similarly, the monsters will be interested in the current position of the `Player` and so will need access to this information.

Consequently, in addition to `Load()` and `Run()`, class `Dungeon` will need many other member functions in its public interface – functions like "Accessible()", and "Remove Monster()". The full set of member functions will get sorted out steadily as different aspects of the game are considered in detail.

Most "dungeon items" will need to interact with the `Dungeon` object in some way or other. It would be best if they all have a `Dungeon*` data member that gets initialized as they are constructed.

class Player

The `Player` object's main responsibility will be getting and interpreting a command entered by the user.

Movement commands

Commands are input as single characters and define either movements or, in this game, directional applications of destructive "magic". The characters 1…9 can be used to define movements. If the keyboard includes a number pad, the convenient mapping is 7 = "north west", 8 = "north", 9 = "north east", 4 = "west" and so forth (where "north" means movement toward the top of the screen and "west" means movement leftwards). Command 5 means "no movement" (sometimes a user may want to delay a little, e.g. to let the player object recover from injury).

Magic action commands

The "magic action" commands can use the keys q, w, e, a, d, z, x, and c (on a standard QWERTY keyboard, these have the same relative layout and hence define the same directional patterns as the keys on the numeric keypad).

Player::Run()

The main `Player::Run()` function will be something like:

```
Player::Run()
    char ch = GetUserCommand();
    if(isdigit(ch)) PerformMovementCommand(ch);
    else PerformMagicCommand(ch);
    UpdateState();
    ShowStatus();
```

It will involve several auxiliary (private) member functions of class `Player`.

Auxiliary private member functions used by Run()

A `GetUserCommand()` function can arrange to read the input. Input is echoed at the current location of the cursor. This could mess up the map display. Consequently it will be necessary to position the cursor prior to reading a command character. This work of cursor positioning and actual data input will involve interactions with window object(s).

A function `UpdateState()` can deal with the business about a `Player` object's health and manna levels increasing. A `ShowStatus()` function can keep the displays current; again this will involve interactions with windows.

The Perform… functions will involve interactions with the `Dungeon` object, and possibly other objects as well.

class Collectable

The collectable items could be made instances of a class `Collectable`. It does not seem that there will be any differences in their behaviours, so there probably won't be any specialized subclasses of class `Collectable`. At this stage, it doesn't appear as if `Collectable` objects will do much at all.

They have to draw themselves when asked (presumably by the `Dungeon` object when it is organizing displays); they will own a character that they use to represent themselves. They will also need to own integer values representing the amounts by which they change the `Player` object's health etc when they get taken. Some access functions will have to exist so that the `Player` object can ask for the relevant data.

A monster object moving onto the same point as a `Collectable` object will hide it. When the monster object moves away, the `Collectable` object should be redrawn. The `Dungeon` object had better arrange to get all `Collectable` objects draw themselves at regular intervals; this code could be added to the `while()` loop in `Dungeon::Run()`.

class Monster

As explained in the dungeon game specification, the basic behaviour of a `Monster` is to attack whenever possible, otherwise to advance toward the `Player` when this is possible, otherwise to continue with some "normal action". This behaviour could be defined in the `Monster::Run()` function which would involve a number of auxiliary functions:

Monster::Run()

```
Monster::Run()
    if(CanAttack())
```

```
        Attack();
else
if(CanDetect())
        Advance();
else
NormalMove();
```

Different subclasses of class `Monster` can specialize the auxiliary functions so as to vary the basic behaviour. Naturally, these functions will be declared as `virtual`.

Auxiliary private member functions used by Run()

Default definitions are possible for some member functions. The default `CanAttack()` function should return true if the `Player` object is adjacent. The default `Attack()` function would tell the `Player` object that it has been hit for a specified number of points of damage. The default implementations for the other functions could all be "do nothing" (i.e. just an empty body `{ }` for `Advance()` and `NormalMove()` and a `return 0` for `CanDetect()`).

Checking adjacency will involve getting a pointer to the `Player` object (this can be provided by the `Dungeon` object) and then asking the `Player` for its position. It might be worth having some simple class to represent (x, y) point coordinates. A `Monster` object could have an instance of class `Pt` to represent its position. The `Player` could return its coordinates as an instance of class `Pt`. The first `Pt` could be asked whether it is adjacent to the second.

29.2.2 WindowRep and Window classes

Previous experience with practical windowing systems has influenced the approach developed here for handling the display. As illustrated in Figure 29.2, the display system uses class `WindowRep` and class `Window` (and its specialized subclasses).

WindowRep

Actual communication with the screen is the responsibility of a class `WindowRep` (Window Representation). Class `WindowRep` encapsulates all the sordid details of how to talk to an addressable screen (using those obscure functions, introduced in Chapter 12, like `cgotoxy(x,y,stdout);`). In addition, it is responsible for trying to optimize output to the screen. When the `WindowRep` object gets a request to output a character at a specific point on the screen, it only performs an output operation if the character given is different from that already shown. In order to do this check, the `WindowRep` object maintains a character array in memory that duplicates the information currently on the screen.

The program will have exactly one instance of class `WindowRep`. All "window" objects (or other objects) that want to output (or input) a character will have to interact with the `WindowRep` object. (There can only be one `WindowRep` object in a program because there is only one screen and this screen has to have a unique owner that maintains consistency etc.)

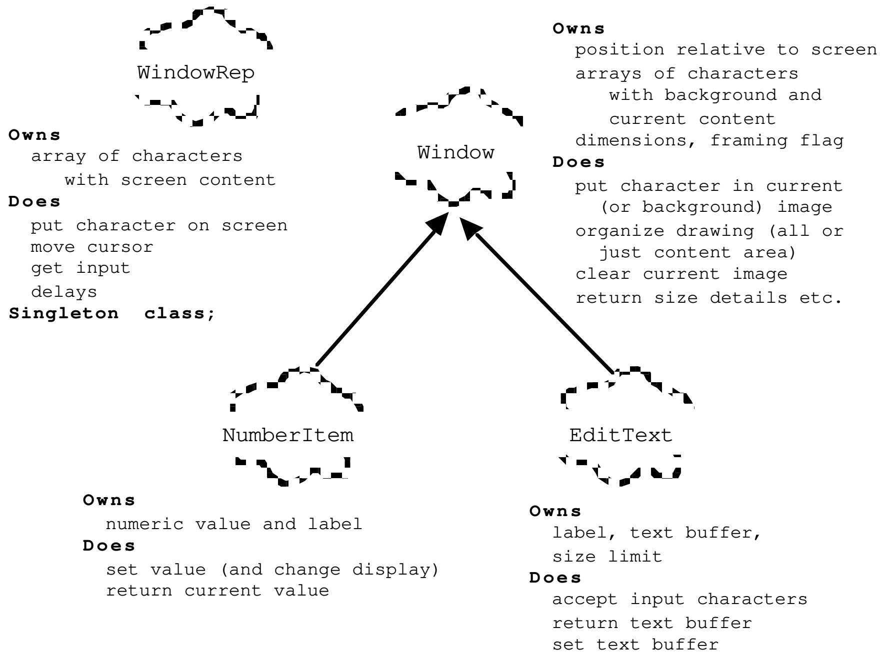

Figure 29.2 Class WindowRep and the Windows class hierarchy.

"Singleton" pattern

A class for which there can only be a single instance, an instance that must be accessible to many other objects, an instance that may have to create auxiliary data structures or perform some specialized hardware initialization – this is a common pattern in programs. A special term "singleton class" has been coined to describe this pattern. There are some standard programming "cliches" used when coding such classes, they are followed in this implementation.

The unique `WindowRep` object used in a program provides the following services:

- PutCharacter
  Outputs a character at a point specified in screen coordinates.

- MoveCursor
  Positions the "cursor" prior to an input operation.

- GetChar
  Inputs a character, echoing it at the current cursor position

- Clear
  Clears the entire screen.

- CloseDown
  Closes down the windowing system and gets rid of the program's `WindowRep` object

There is another specialized static (class) function, `Instance()`. This handles aspects of the "singleton pattern" (programming cliche) as explained in the implementation (Section 29.3). Essentially, the job of this function is to make certain that there is an instance of class `WindowRep` that is globally available to any other object that needs it (if necessary, creating the program's unique `WindowRep` object).

Window

Window objects, instances of class `Window` or its subclasses, are meant to be things that own some displayable data (an array of characters) and that can be "mapped onto the screen". A `Window` object will have coordinates that specify where its "top-left" corner is located relative to the screen. (In keeping with most cursor-addressable screen usage, coordinate systems are 1-based rather than 0-based so the top left corner of the screen is at coordinate (1,1).) `Window` objects also define a size in terms of horizontal and vertical dimensions. Most `Window` objects are "framed", their perimeters are marked out by '-', '|', and '+' characters (as in Figure 29.1). A `Window` may not fit entirely on the screen (the fit obviously depends on the size and the origin). The `WindowRep` object resolves this by ignoring output requests that are "off screen".

Background and current (foreground) window contents

`Window` objects have their own character arrays for their displayable content. Actually, they have two character arrays: a "background array" and a "current array". When told to "draw" itself, a `Window` object executes a function involving a double loop that takes characters from the "current array", works out where each should be located in terms of screen coordinates (taking into account the position of the `Window` object's top-left corner) and requests the `WindowRep` object to display the character at the appropriate point.

The "background array" defines the initial contents of the window (possibly all blank). Before a window is first shown on the screen, its current array is filled from the background. A subsequent "clear" operation on a specific point in the window, resets the contents of the current window to be the same as the background at the specified point.

A specific background pattern can be loaded into a window by setting the characters at each individual position. In the dungeon game, the `Dungeon` object owns the window used to display the map; it sets the background pattern for that window to be the same as its map layout.

What does a Window do?

The `Window` class has the following public functions:

- Constructor
  Sets the size and position fields; creates arrays.

- Destructor
  Gets rid of arrays. (The destructor is `virtual` because class `Window` is to serve as the base class of a hierarchy. In class hierarchies, base classes must always define virtual destructors.)

- Set, Clear
  Change the character at a single point in the current (foreground) array.

- SetBkgd
  Change the character at a single point in the background array.

- Access functions: X, Y, Width, Height
  Return details of data members.

- PrepareContent
  Initialize current array with copy of background and, if appropriate, add frame.

- ShowAll, ShowContent
  Output current array via the `WindowRep`.

The class requires a few auxiliary member functions. For example, the coordinates passed to functions like `Set()` must be validated.

What does a Window own?

A `Window` owns its dimension data and its arrays. These data members should be `protected`; subclasses will require access to these data.

Provision for class hierarchy

The functionality of class `Window` will be extended in its subclasses. However the subclasses don't change the existing functions like `ShowAll()`. Consequently, these functions are not declared as virtual.

"Inheritance for extension" and "inheritance for redefinition"

The relationships between class `Window` and its subclasses, and class `Monster` and its subclasses, are subtly different. The subclasses of `Window` *add* functionality to a working class, but don't change its basic behaviours. Consequently, the member functions of class `Window` are non-virtual (apart from the destructor). Class `Monster` defines a general abstraction; e.g. all `Monster` object can execute some "NormalMove" function, different subclasses redefine the meaning of "NormalMove". Many of the member functions of class `Monster` are declared as virtual so as to permit such redefinition. Apart from the differences with respect to the base class member function being virtual or non-virtual, you will also see differences in the accessibility of additional functions defined by subclasses. When inheritance is being used to extend a base class, many of the new member functions appear in the public interface of the subclass. When inheritance is being used to specialize an existing base class, most of the new functions will be private functions needed to implement changes to some existing behaviour. Both styles, "inheritance for extension" and "inheritance for redefinition", are common.

Subclasses of class Window

This program has two subclasses for class `Window`: `NumberItem` and `EditText`. Instances of class `NumberItem` are used to display numeric values; instances of class `EditText` can be used to input commands. As illustrated in Figure 29.3, these are displayed as framed windows that are 3 rows deep by *n*-columns wide. The left part of the window will normally contain a textual label. The right part is used to display a string representing a (signed) numeric value or as input field where input characters get echoed (in addition to being stored in some data buffer owned by the object).

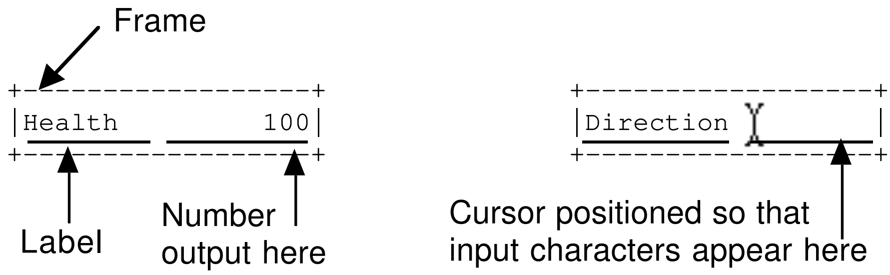

Figure 29.3 NumberItem and EditText Windows

<u>NumberItem</u>

The class declaration for `NumberItem` is:

```
class NumberItem : public Window {
public:
    NumberItem(int x, int y, int width, char *label,
                           long initval = 0);
    void    SetVal(long newVal);
    long    GetVal() { return fVal; }
private:
    void    SetLabel(int s, char*);
    void    ShowValue();
    long    fVal;
    int     fLabelWidth;
};
```

What does a NumberItem own?

In addition to the character arrays and dimension data that a `NumberItem` object already has because it is a kind of `Window`, a `NumberItem` owns a long integer holding the value to be displayed (and, also, an integer to record the number of characters used by the label so that numeric outputs don't overwrite the label).

What does a NumberItem do?

The constructor for class `NumberItem` completes the normal initialization processes of class `Window`. The auxiliary private member function `SetLabel()` is used to copy the given label string into the background array. The inherited `PrepareContent()` function loads the current array from the background and adds the frame. Finally, using the auxiliary `ShowValue()` function, the initial number is converted into characters that are added to the current image array.

Once constructed, a `NumberItem` can be used by the program. Usually, usage will be restricted to just three functions – `GetVal()` (`return fVal;`), `SetVal()` (changes `fVal` and uses `ShowValue()` to update the current image array), and `ShowAll()` (the function, inherited from class `Window`, that gets the window displayed).

EditText

The primary responsibility of an `EditText` object is getting character input from the user. In the dungeon game program, individual input characters are required as command characters. More generally, the input could be a multicharacter word or a complete text phrase.

An `EditText` object should be asked to get an input. It should be responsible for getting characters from the `WindowRep` object (while moving the cursor around to try to get the echoed characters to appear at a suitable point on the screen). Input characters should be added to a buffer maintained by the `EditText` object. This input process should terminate when either a specified number of characters has been received or a recognizably distinct terminating character (e.g. 'tab', 'enter') is input. (The dungeon program can use such a more generalized `EditText` object by specifying that the input operation is to terminate when a single character has been entered). Often, the calling program will need to know what character terminated the input (or whether it was terminated by a character limit being reached). The input routine can return the terminating character (a '\0' could be returned to indicate that a character count limit was reached).

The `EditText` class would have to provide an access function that lets a caller read the characters in its buffer.

The class declaration for class `EditText` is:

```
class EditText: public Window {
public:
    EditText(int x, int y, int width, char *label, short size);
    void   SetVal(char*);
    char   *GetVal() { return fBuf; }
    char   GetInput();
private:
    void   SetLabel(int s, char*);
    void   ShowValue();
    int    fLabelWidth;
    char   fBuf[256];
    int    fSize;
    int    fEntry;
};
```

What does a EditText own?

In addition to the data members inherited from class `Window`, a `EditText` owns a (large) character buffer that can be used to store the input string, integers to record the number of characters entered so far and the limit number. Like the `NumberItem`, an `EditText` will also need a record of the width of its label so that the input field and the label can be kept from overlapping.

What does a NumberItem do?

The constructor for class `EditText` completes the normal initialization processes of class `Window`. The auxiliary private member function `SetLabel()` is used to copy the given label string into the background array. The inherited `PrepareContent()` function loads the current array from the background and adds the frame. The buffer, `fBuf`, can be "cleared" (by setting `fBuf[0]` to '\0')..

The only member functions used in most programs would be `GetInput()`, `GetVal()` and `ShowAll()`. Sometimes, a program might want to set an initial text string (e.g. a prompt) in the editable field (function `SetVal()`).

29.2.3 DungeonItem hierarchy

Class `Monster` is meant to be an abstraction; the real inhabitants of the dungeon are instances of specialized subclasses of class `Monster`.

Class `Monster` has to provide the following functions (there may be others functions, and the work done in these functions may get expanded later, this list represents an initial guess):

- Constructor and destructor
  The constructor will set a `Dungeon*` pointer to link back to the `Dungeon` object and set a char data member to the symbol used to represent the `Monster` on the map view.
  Since class `Monster` is to be in a hierarchy, it had better define a virtual destructor.

- Read
  A `Monster` is supposed to read details of its initial position, its "health" and "strength" from an input file. It will need data members to store this information.

- Access functions to get position, to check whether "alive", …

- A `Run()` function that as already outlined will work through redefinable auxiliary functions like `CanAttack()`, `Attack()`, `CanDetect()`, `Advance()` and `NormalMove()`.

- Draw and Erase functions.

- A Move function.
  Calls `Erase()`, changes coords to new coords given as argument, and calls `Draw()`.

- GetHit function
  Reduces "health" attribute in accord with damage inflicted by `Player`.

The example implementation has three specializations: `Ghost`, `Patrol`, and `Wanderer`. These classes redefine member functions from class `Monster` as needed.

class Ghost

A `Ghost` is a `Monster` that:

- uses the default "do nothing" implementation defined by `Monster::NormalMove()` along with the standard `CanAttack()`, `Attack()` functions;

- has a `CanDetect()` function that returns true when the player is within a fixed distance of the point where the `Ghost` is located (the presence of intervening walls makes no difference to a `Ghost` object's power of detection);

- has an `Advance()` function that moves the `Ghost` one square vertically, diagonally, or horizontally so as to advance directly toward the player; a `Ghost` can move through walls;
- has a high initial "health" rating;
- inflicts only a small amount of damage when attacking the `Player`.

Class `Ghost` needs to redefine only the `Advance()` and `CanDetect()` functions. Since a `Ghost does` not require any additional data it does not to change the `Read()` function.

class Patrol A `Patrol` is a `Monster` that:

- uses the default `CanAttack()`, `Attack()` functions to attack an adjacent player;
- has a `CanDetect()` function that returns true there is a clear line of sight between it and the `Player` object;
- has an `Advance()` function that instead of moving it toward the `Player` allows it to fire a projectile that follows the "line of sight" path until it hits the `Player` (causing a small amount of damage), the movement of the projectile should appear on the screen;
- has a `NormalMove()` function that causes it to follow a predefined patrol route (it never departs from this route so it does not attempt to pursue the `Player`).
- has a moderate initial "health" rating;
- inflicts a large amount of damage when making a direct attack on an adjacent player.

The patrol route should be defined as a sequence of points. These will have to be read from the input file and so class `Patrol` will need to extend the `Monster::Read()` function. The `Patrol::Read()` function should check that the given points are adjacent and that all are accessible (within the bounds of the dungeon and not blocked by walls).

Class `Patrol` will need to define extra data members to hold the route data. It will need an array of `Pt` objects (this can be a fixed sized array with some reasonable maximum length for a patrol route), an integer specifying the number of points in the actual route, an integer (index value) specifying which `Pt` in the array the `Patrol` is currently at, and another integer to define the direction that the `Patrol` is walking. (The starting point given for the `Patrol` will be the first element of the array. Its initial moves will cause it to move to successive elements of the `Pt` array; when it reaches the last, it can retrace its path by having the index decrease through the array.)

class Wanderer A `Wanderer` is a `Monster` that:

- uses the default `CanAttack()`, `Attack()` functions to attack an adjacent player;

- has a `CanDetect()` function that returns true there is a clear line of sight between it and the `Player` object;
- has an `Advance()` function that causes the `Wanderer` to move one step along the line of sight path toward the current position of the `Player`;
- has a `NormalMove()` function that causes it to try to move in a constant direction until blocked by a wall, when its movement is blocked, it picks a new direction at random;
- has a small initial "health" rating;
- inflicts a moderate amount of damage when making a direct attack on an adjacent player.

A `Wanderer` will need to remember its current direction of movement so that it can keep trying to go in the same direction. Integer data members, representing the current delta-x, delta-y changes of coordinate, could be used.

Commonalities between class Player and class Monster

There are similarities between the `Monster` and `Player` classes. Both classes define things that have a `Pt` coordinate, a health attribute and a strength attribute. There are similarities in behaviours: both the `Player` and the `Monsters` read initial data from file, get told that they have been hit, get asked whether they are still alive, get told to draw themselves (and erase themselves), have a "move" behaviour that involves erasing their current display, changing their `Pt` coordinate and then redrawing themselves. These commonalities are sufficient to make it worth defining a new abstraction, "active item", that subsumes both the `Player` and the `Monsters`.

This process of abstracting out commonalities can be repeated. There are similarities between class Collectable and the new class `ActiveItem`. Both are things with `Pt` coordinates, draw and erase behaviours; they respond to queries about where they are (returning their `Pt` coordinate). These common behaviours can be defined in a base class: `DungeonItem`.

The `DungeonItem` class hierarchy used in the implementation is shown in Figure 29.4.

29.2.3 Finalising the classes

Completion of the design stage involves coming up with the class declarations of all the classes, possibly diagramming some of the more complex patterns of interaction among instances of different classes, and developing algorithms for any complicated functions.

Class Dungeon

The finalised declaration for class `Dungeon` is:

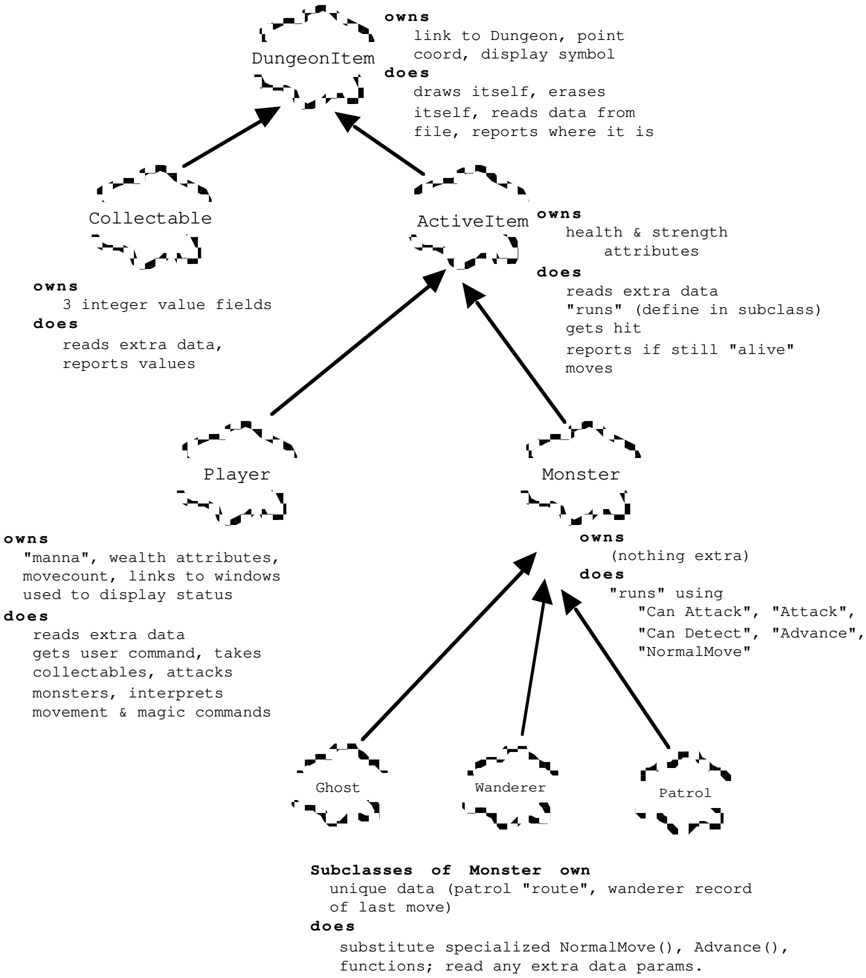

Figure 29.4 DungeonItem class hierarchy.

```
class Dungeon {
public:
    Dungeon();
    ~Dungeon();
    void   Load(const char filename[]);
    int    Run();
    int    Accessible(Pt p) const;
    Window *Display();
    Player *Human();

    int    ValidPoint(Pt p) const;

    Monster *M_at_Pt(Pt p);
    Collectable *PI_at_Pt(Pt p);
    void   RemoveProp(Collectable *pi);
    void   RemoveM(Monster *m);
```

```
    int     ClearLineOfSight(Pt p1, Pt p2, int max,
                   Pt path[]);

private:
    int     ClearRow(Pt p1, Pt p2, int max, Pt path[]);
    int     ClearColumn(Pt p1, Pt p2, int max, Pt path[]);
    int     ClearSemiVertical(Pt p1, Pt p2, int max,
                          Pt path[]);
    int     ClearSemiHorizontal(Pt p1, Pt p2, int max,
                          Pt path[]);
    void    LoadMap(ifstream& in);
    void    PopulateDungeon(ifstream& in);
    void    CreateWindow();

    DynamicArray  fProps;
    DynamicArray  fInhabitants;
    Player        *fPlayer;

    char          fDRep[MAXHEIGHT][MAXWIDTH];
    Window        *fDWindow;
    int           fHeight;
    int           fWidth;
};
```

What does a Dungeon own?

The `Dungeon` object owns the map of the maze (represented by its data elements `fDRep[][]`, `fHeight`, and `fWidth`). It also owns the main map window (`fDWindow`), the `Player` object (`fPlayer`) and the collections of `Monsters` and `Collectables`. Data members that are instances of class `DynamicArray` are used for the collections (`fInhabitants` for the `Monsters`, `fProps` for the `Collectables`).

What does a Dungeon do?

The `Load()` and `Run()` functions are used by the main program. Function `Load()` makes uses of the auxiliary private member functions `LoadMap()` and `PopulateDungeon()`; these read the various data and create `Monster`, `Collectable`, and `Player` object(s) as specified by the input. The auxiliary private member function `CreateWindow()` is called from `Run()`; it creates the main window used for the map and sets its background from information in the map.

Access functions like `Display()` and `Human()` allow other objects to get pointers to the main window and the Player object. The `ActiveItem` objects that move are going to need access to the main Window so as to tell it to clear and set the character that is to appear at a particular point.

The `ValidPoint()` function checks whether a given Pt is within the bounds of the maze and is not a "wall".

The functions `M_at_Pt()` and `PI_at_Pt()` involve searches through the collections of `Monsters` and `Collectables` respectively. These function return the first member of the collection present at a `Pt` (or `NULL` if there are no objects at that `Pt`). The Remove… function eliminate members of the collections.

Class `Dungeon` has been given responsibility for checking whether a "clear line of sight" exists between two `Pts` (this function is called in both `Wanderer::CanDetect()` and `Patrol::CanDetect()`). The function takes as arguments the two points, a maximum range and a `Pt` array in which to return the `Pts` along the line of sight. Its implementation uses the auxiliary private member functions `ClearRow()` etc.

The algorithm for the `ClearLineOfSight()` function is the most complex in the program. There are two easy cases; these occur when the two points are in the same row or the same column. In such cases, it is sufficient to check each point from start to end (or out to a specified maximum) making certain that the line of sight is not blocked by a wall. Pseudo code for the `ClearRow()` function is:

```
Dungeon::ClearRow(Pt p1, Pt p2, int max, Pt path[])
    delta = if p1 left of p2 then 1 else -1
    current point = p1
    for i < max do
           current point's x  += delta;
           if(current point is not accessible) return fail
           path[i] = current point;
           if(current point equal p2) return success
           i++
    return fail
```

Cases where the line is oblique are a bit more difficult. It is necessary to check the squares on the map (or screen) that would be traversed by the best approximation to a straight line between the points. There is a standard approach to solving this problem; Figure 29.5 illustrates the principle.

The squares shown by dotted lines represent the character grid of the map or screen; they are centred on points defined by integer coordinates. The start point and end point are defined by integer coordinates. The real line between the points has to be approximated by a sequence of segments joining points defined by integer coordinates. These points define which grid squares are involved. In Figure 29.5 the squares traversed are highlighted by • marks.

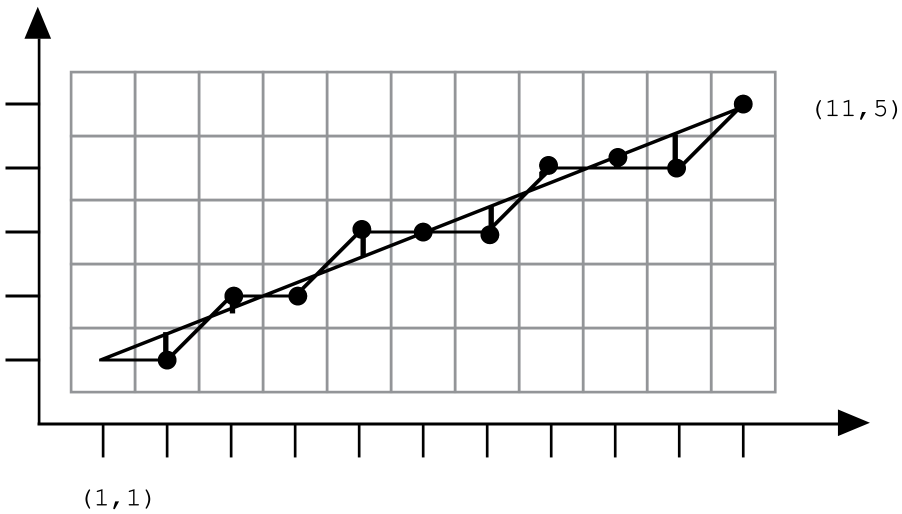

Figure 29.5 A digitized line.

The algorithm has to chose the best sequence of integer points, for example choosing point (2, 1) rather than (2, 2) and (8, 4) rather than (8, 3) or (8, 5). The algorithm works by calculating the error associated with different points (shown by the vertical lines in Figure 29.5). The correct y value for say x=8 can be calculated; the errors of taking y = 3, or 4, or 5 are easy to calculate and the best approximation is chosen. When the squares traversed have been identified, they can be checked to determine that they are not blocked by walls and, if clear, added to the `path` array.

The algorithm is easiest to implement using two separate versions for the cases where the change in x is larger or the change in y is larger. A pseudo-code outline for the algorithm for where the change in x is larger is as follows:

```
Dungeon::ClearSemiHorizontal(Pt p1, Pt p2, int max,
     Pt path[])
    ychange = difference in y values of two points
    xchange = difference in x values of two points
    if(xchange > max)
           return fail

    deltax = if x increasing then 1 else -1
    deltay = if y increasing then 1 else -1

    slope = change in y divided by change in x
    error = slope*deltax

    current point = p1
    for i < abs(xchange) do
           if(error*deltay>0.5)                                Pick best next point
                  current point y += deltay
                  error -= deltay

           error += slope*deltax
           current point x += deltax
           if(current point not accessible) return fail        Check accessibility

           path[i] = current point                             Add to path
           if(current point equal p2) return success
           i++
    return fail
```

Class Pt

It is worth introducing a simple class to represent coordinates because many of the functions need coordinates as arguments, and there are also several places in the code where it is necessary to compare the coordinates of different objects. A declaration for class `Pt` is:

```
class Pt {
public:
    Pt(int x = 0, int y = 0);
    int    X() const;
    int    Y()    const;
    void   SetPt(int newx, int newy);
```

```
        void    SetPt(const Pt& other);
        int     Equals(const Pt& other) const;
        int     Adjacent(const Pt& other) const;
        int     Distance(const Pt& other) const;
    private:
        int     fx;
        int     fy;
    };
```

Most of the member functions of `Pt` are sufficiently simple that they can be defined as `inline` functions.

Classes WindowRep and Window

The declaration for class WindowRep is

```
    class WindowRep {
    public:
        static WindowRep *Instance();
        void    CloseDown();
        void    PutCharacter(char ch, int x, int y);
        void    Clear();
        void    Delay(int seconds) const;
        char    GetChar();
        void    MoveCursor(int x, int y);
    private:
        WindowRep();
        void    Initialize();
        void    PutCharacter(char ch);
        static WindowRep *sWindowRep;
        char    fImage[CG_HEIGHT][CG_WIDTH];
    };
```

The size of the image array is defined by constants `CG_HEIGHT` and `CG_WIDTH` (their values are determined by the area of the cursor addressable screen, typically up to 24 high, 80 wide).

The static member function `Instance()`, and the static variable `sWindowRep` are used in the implementation of the "singleton" pattern as explained in the implementation section. Another characteristic of the singleton nature is the fact that the constructor is private; a `WindowRep` object can only get created via the public `Instance()` function.

Most of the members of class `Window` have already been explained. The actual class declaration is:

```
    class Window {
    public:
        Window(int x, int y, int width, int height,
                char bkgd = ' ', int framed = 1);
        virtual         ~Window();
        void    Clear(int x, int y);
        void    Set(int x, int y, char ch);
        void    SetBkgd(int x, int y, char ch);
```

```
    int     X() const;
    int     Y() const;
    int     Width() const;
    int     Height() const;

    void    ShowAll() const;
    void    ShowContent() const;
    void    PrepareContent();
protected:
    void    Change(int x, int y, char ch);
    int     Valid(int x, int y) const;
    char    Get(int x, int y, char **img) const;
    void    SetFrame();
    char    **fBkgd;
    char    **fCurrentImg;
    int     fX;
    int     fY;
    int     fWidth;
    int     fHeight;
    int     fFramed;
};
```

The declarations for the specialized subclasses of class `Window` were given earlier.

DungeonItem class hierarchy

The base class for the hierarchy defines a `DungeonItem` as something that can read its data from an input stream, can draw and erase itself, and can say where it is. It owns a link to the `Dungeon` object, its `Pt` coordinate and a symbol.

DungeonItem

```
class DungeonItem {
public:
    DungeonItem(Dungeon *d, char sym);
    virtual ~DungeonItem();
    Pt              Where() const;
    virtual void    Draw();
    virtual void    Read(ifstream& in);
    virtual void    Erase();
protected:
    Dungeon         *fD;
    Pt              fPos;
    char            fSym;
};
```

Since class `DungeonItem` is the base class in a hierarchy, it provides a virtual destructor and makes its data members protected (allowing access by subclasses).

A `Collectable` object is just a `DungeonItem` with three extra integer data members and associated access functions that can return their values. Because a `Collectable` needs to read the values of its extra data members, the class redefines the `DungeonItem` read function.

Collectable

```
class Collectable : public DungeonItem {
public:
    Collectable(Dungeon* d, char sym);
    int           Hlth();
    int           Wlth();
    int           Manna();
    virtual void  Read(ifstream& in);
private:
    int           fHval;
    int           fWval;
    int           fMval;
};
```

An `ActiveItem` is a `DungeonItem` that gets hit, gets asked if is alive, moves, and runs. Member function `Run()` is pure virtual, it has to be redefined in subclasses (because the "run" behaviours of subclasses `Player` and `Monster` are quite different). Default definitions can be provided for the other member functions. All `ActiveItem` objects have "strength" and "health" attributes. The inherited `DungeonItem::Read()` function will have to be extended to read these extra data.

ActiveItem

```
class ActiveItem : public DungeonItem {
public:
    ActiveItem(Dungeon *d, char sym);
    virtual void  Read(ifstream& in);
    virtual void  Run() = 0;
    virtual void  GetHit(int damage);
    virtual int   Alive() const;
protected:
    virtual void  Move(const Pt& newpoint);
    Pt      Step(int dir);
    int     fHealth;
    int     fStrength;
};
```

(Function `Step()` got added during implementation. It returns the coordinate of the adjacent point as defined by the `dir` parameter; 7 => north west neighbor, 8 => north neighbor etc.)

A `Player` is a specialized `ActiveItem`. The class has one extra public member function, `ShowStatus()`, and several additional private member functions that are used in the implementation of its definition of `Run()`. A `Player` has extra data members for its wealth and manna attributes, a move counter that gets used when updating health and manna. The `Player` object owns the `NumberItem` and `EditText` windows that are used to display its status and get input commands.

Player

```
class Player : public ActiveItem {
public:
    Player(Dungeon* d);
    virtual void  Run();
    virtual void  Read(ifstream& in);
    void    ShowStatus();
private:
    void    TryMove(int newx, int newy);
```

```
    void    Attack(Monster *m);
    void    Take(Collectable *pi);
    void    UpdateState();
    char    GetUserCommand();
    void    PerformMovementCommand(char ch);
    void    PerformMagicCommand(char ch);

    int            fMoveCount;
    int            fWealth;
    int            fManna;
    NumberItem     *fWinH;
    NumberItem     *fWinW;
    NumberItem     *fWinM;
    EditText       *fWinE;
};
```

Class `Monster` is just an `ActiveItem` with a specific implementation of `Run()` that involves the extra auxiliary functions `CanAttack()` etc. Since instances of specialized subclasses are going to be accessed via `Monster*` pointers, and there will be code of the form `Monster *m; … ; delete m;`, the class should define a virtual destructor.

Monster

```
class Monster : public ActiveItem {
public:
    Monster(Dungeon* d, char sym);
    virtual ~Monster();
    virtual void  Run();
protected:
    virtual int CanAttack();
    virtual void Attack();
    virtual int CanDetect();
    virtual void Advance();
    virtual void NormalMove() { }
};
```

Class `Ghost` defines the simplest specialization of class `Monster`. It has no extra data members. It just redefines the default (do nothing) `CanDetect()` and `Advance()` member functions.

Ghost

```
class Ghost : public Monster {
public:
    Ghost(Dungeon *d);
protected:
    virtual int CanDetect();
    virtual void Advance();
};
```

The other two specialized subclasses of `Monster` have additional data members. In the case of class `Patrol`, these must be initialized from input so the `Read()` function is redefined. Both classes redefine `Normal Move()` as well as `CanDetect()` and `Advance()`.

```
class Wanderer : public Monster {
public:
    Wanderer(Dungeon *d);
protected:
    virtual void  NormalMove();
    virtual int CanDetect();
    virtual void Advance();
    int     fLastX, fLastY;
    Pt      fPath[20];
};

class Patrol : public Monster {
public:
    Patrol(Dungeon *d);
    virtual void  Read(ifstream& in);
protected:
    virtual void  NormalMove();
    virtual int CanDetect();
    virtual void Advance();
    Pt      fPath[20];
    Pt      fRoute[100];
    int     fRouteLen;
    int     fNdx, fDelta;
};
```

Object Interactions

Figure 29.6 illustrates some of the interactions involved among different objects during a cycle of `Dungeon::Run()`.

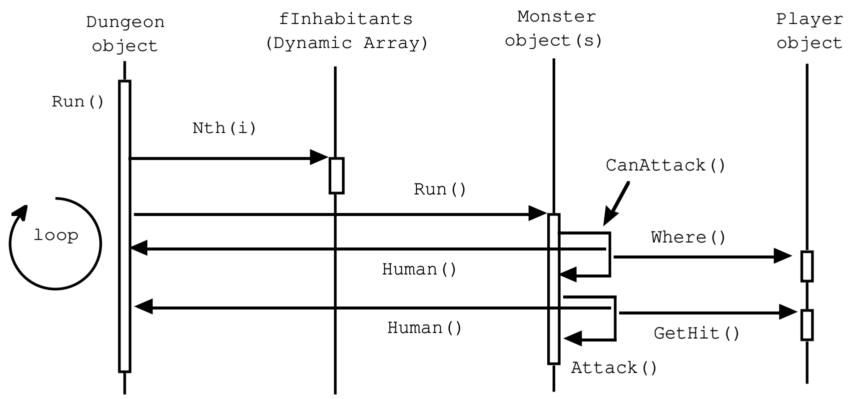

Figure 29.6 Some of the object interactions in Dungeon::Run.

The diagram illustrates aspects of the loop where each `Monster` object in the `fInhabitants` collection is given a chance to run. The `Dungeon` object will first interact with the `DynamicArray fInhabitants` to get a pointer to a particular `Monster`. This will then be told to run; its `Run()` function would call its `CanAttack()` function.

In CanAttack(), the Monster would have to get details of the Player object's position. This would involve first a call to member function Human() of the Dungeon object to get a pointer, and then a call to Where() of the Player object.

The diagram in Figure 29.7 illustrates the case where the Player is adjacent and the Monster object's Attack() function is called. This will again involve a call to Dungeon::Human(), and then a call to the GetHit() function of the Player object.

Figure 29.7 illustrates some of the interactions that might occur when the a movement command is given to Player:Run().

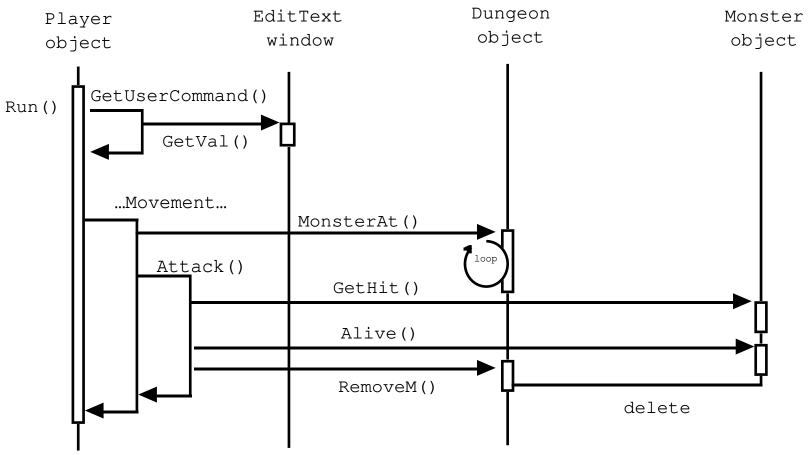

Figure 29.7 Object interactions during Player::Run.

The Player object would have had to start by asking its EditText window to get input. When EditText::GetInput() returns, the Player object could inspect the character string entered (a call to the EditText's GetVal() function, not shown in diagram). If the character entered was a digit, the Player object's function PerformMovementCommand would be invoked. This would use Step() to determine the coordinates of the adjacent Pt where the Player object was to move. The Player would have to interact with the Dungeon object to check whether the destination point was occupied by a Monster (or a Collectable).

The diagram in Figure 29.7 illustrates the case where there is an adjacent Monster. The Player object informs the Monster that it has been hit. Then it checks whether the Monster is still alive. In the illustration, the Monster object has been destroyed, so the Player must again interact with the Dungeon object. This removes the Monster from the fInhabitants list (interaction with the DynamicArray is not shown) and deletes the Monster.

29.3 AN IMPLEMENTATION

The files used in the implementation, and their interdependencies are summarized in Figure 29. 8.

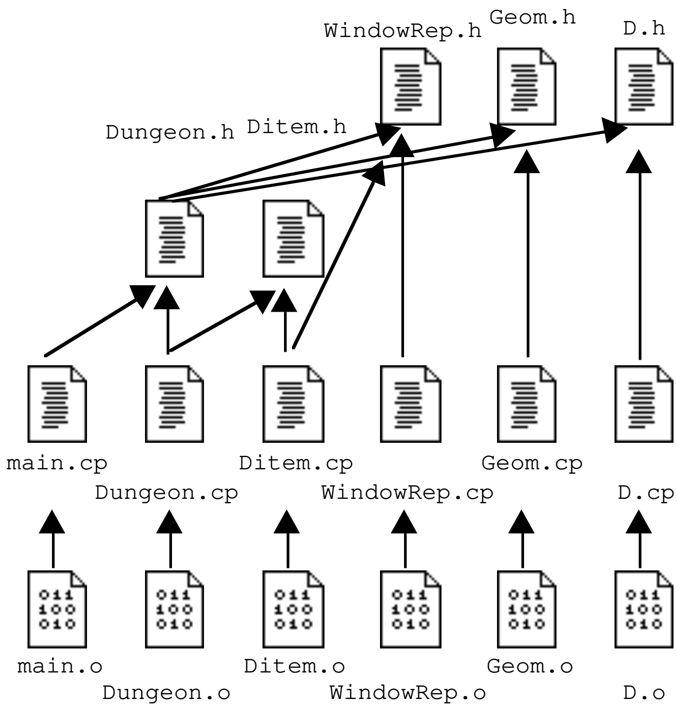

Figure 29.8 Module structure for Dungeon game example.

The files D.h and D.cp contain the `DynamicArray` code defined in Chapter 21. The Geom files have the definition of the simple `Pt` class. The `WindowRep` files contain `WindowRep`, `Window` and its subclasses. DItem.h and DItem.cp contain the declaration and definition of the classes in the `DungeonItem` hierarchy while the Dungeon files contain class `Dungeon`.

An outline for `main()` has already been given; function `Terminate()`, which prints an appropriate "you won" or "you lost" message, is trivial.

29.3.1 Windows classes

There are two aspects to class `WindowRep`: its "singleton" nature, and its interactions with a cursor addressable screen.

The constructor is private. `WindowRep` objects cannot be created by client code (we only want one, so we don't want to allow arbitrary creation). Clients (like the code of class `Dungeon`) always access the unique `WindowRep` object through the static member function `Instance()`.

```
WindowRep *WindowRep::Instance()
{
    if(sWindowRep == NULL)
            sWindowRep = new WindowRep;
```

```
    return sWindowRep;
}

WindowRep *WindowRep::sWindowRep = NULL;
```

The first time that it is called, function `Instance()` creates the `WindowRep` object; subsequent calls always return a pointer to the same object.

The constructor calls function `Initialize()` which performs any system dependent device initialization. It then sets up the image array, and clears the screen.

```
WindowRep::WindowRep()
{
    Initialize();
    for(int row = 0; row < CG_HEIGHT; row++)
           for(int col = 0; col< CG_WIDTH; col++)
                  fImage[row][col] = ' ';
    Clear();
}
```

The implementation of functions like `Initialize()` involves the same system dependent calls as outlined in the "Cursor Graphics" examples in Chapter 12. Some example functions are:

```
void WindowRep::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 IDE
*/
#endif
}

void WindowRep::MoveCursor(int x, int y)
{
    if((x<1) || (x>CG_WIDTH)) return;
    if((y<1) || (y>CG_HEIGHT)) return;

#if defined(SYMANTEC)
    cgotoxy(x,y,stdout);
#else
    gotoxy(x,y);
#endif
}

void WindowRep::PutCharacter(char ch)
{
```

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

Functions like `WindowRep::Delay()` and `WindowRep::GetChar()` similarly repackage code from "Cursor Graphics" example.

The `WindowRep::Putcharacter()` function only does the cursor movement and character output operations when necessary. This function also keeps the `WindowRep` object's image array consistent with the screen.

```
void WindowRep::PutCharacter(char ch, int x, int y)
{
    if((x<1) || (x>CG_WIDTH)) return;
    if((y<1) || (y>CG_HEIGHT)) return;

    if(ch != fImage[y-1][x-1]) {
           MoveCursor(x,y);
           PutCharacter(ch);
           fImage[y-1][x-1] = ch;
           }
}
```

The `CloseDown()` function clears the screen, performs any device specific termination, then after a short delay lets the `WindowRep` object self destruct.

```
void WindowRep::CloseDown()
{
    Clear();
#if defined(SYMANTEC)
    csetmode(C_ECHO, stdin);
#endif
    sWindowRep = NULL;
    Delay(2);
    delete this;
}
```

Window

The constructor for class `Window` initializes the simple data members like the width and height fields. The foreground and background arrays are created. They are vectors, each element of which represents an array of characters (one row of the image).

```
Window::Window(int x, int y, int width, int height,
    char bkgd, int framed )
{
    fX = x-1;
    fY = y-1;
```

```
    fWidth = width;
    fHeight = height;
    fFramed = framed;

    fBkgd = new char* [height];
    fCurrentImg = new char* [height];
    for(int row = 0; row < height; row++) {
            fBkgd[row] = new char[width];
            fCurrentImg[row] = new char[width];
            for(int col = 0; col < width; col++)
                    fBkgd[row][col] = bkgd;
            }
}
```

Naturally, the main task of the destructor is to get rid of the image arrays:

```
Window::~Window()
{
    for(int row = 0; row < fHeight; row++) {
            delete [] fCurrentImg[row];
            delete [] fBkgd[row];
            }
    delete [] fCurrentImg;
    delete [] fBkgd;
}
```

Functions like `Clear()`, and `Set()` rely on auxiliary routines `Valid()` and `Change()` to organize the real work. Function `Valid()` makes certain that the coordinates are within the window's bounds. Function `Change()` is given the coordinates, and the new character. It looks after details like making certain that the window frame is not overwritten (if this is a framed window), arranging for a request to the `WindowRep` object asking for the character to be displayed, and the updating of the array.

```
void Window::Clear(int x, int y)
{
    if(Valid(x,y))
            Change(x,y,Get(x,y,fBkgd));
}

void Window::Set(int x, int y, char ch)
{
    if(Valid(x,y))
            Change(x, y, ch);
}
```

(Function `Change()` has to adjust the x, y values from the 1-based scheme used for referring to screen positions to a 0-based scheme for C array subscripting.)

```
void Window::Change(int x, int y, char ch)
{
    if(fFramed)   {
            if((x == 1) || (x == fWidth)) return;
            if((y == 1) || (y == fHeight)) return;
```

```
        }

    WindowRep::Instance()->PutCharacter(ch, x + fX, y + fY);
    x--;
    y--;
    fCurrentImg[y][x] = ch;
}
```

Note the call to `WindowRep::Instance()`. This returns a `WindowRep*` pointer. The `WindowRep` referenced by this pointer is then asked to output the character at the specified point as offset by the origin of this window.

Function `SetBkgd()` simply validates the coordinate arguments and then sets a character in the background array. Function `Get()` returns the character at a particular point in either background of foreground array (an example of its use is in the statement `Get(x, y, fBkgd)` in `Window::Clear()`).

```
char Window::Get(int x, int y, char **img) const
{
    x--;
    y--;
    return img[y][x];
}
```

Function Prepare`Content()` loads the current image array from the background and, if appropriate, calls `SetFrame()` to add a frame.

```
void Window::PrepareContent()
{
    for(int row = 0; row < fHeight; row++)
            for(int col = 0; col < fWidth; col++)
                    fCurrentImg[row][col] = fBkgd[row][col];
    if(fFramed)
            SetFrame();
}

void Window::SetFrame()
{
    for(int x=1; x<fWidth-1; x++) {
            fCurrentImg[0][x] = '-';
            fCurrentImg[fHeight-1][x] = '-';
            }
    for(int y=1; y < fHeight-1; y++) {
            fCurrentImg[y][0] = '|';
            fCurrentImg[y][fWidth-1] = '|';
            }
    fCurrentImg[0][0] = '+';
    fCurrentImg[0][fWidth-1] = '+';
    fCurrentImg[fHeight-1][0] = '+';
    fCurrentImg[fHeight-1][fWidth-1] = '+';
}
```

A `Window` object's frame uses its top and bottom rows and leftmost and rightmost columns. The content area, e.g. the map in the dungeon game, cannot use these perimeter points. (The input file for the map could define the perimeter as all "wall".)

The access functions like `X()`, `Y()`, `Height()` etc are all trivial, e.g.:

```
int Window::X() const
{
    return fX;
}
```

The functions `ShowAll()` and `ShowContent()` are similar. They have loops take characters from the current image and send the to the WindowRep object for display. The only difference between the functions is in the loop limits; function `ShowContent()` does not display the periphery of a framed window.

```
void Window::ShowAll() const
{
    for(int row=1;row<=fHeight; row++)
            for(int col = 1; col <= fWidth; col++)
                   WindowRep::Instance()->
                          PutCharacter(
                                 fCurrentImg[row-1][col-1],
                                 fX+col, fY+row);
}
```

NumberItem and EditText

The only complications in class `NumberItem` involve making certain that the numeric value output does not overlap with the label. The constructor checks the length of the label given and essentially discards it if display of the label would use too much of the width of the `NumberItem`.

```
NumberItem::NumberItem(int x, int y, int width, char *label,
    long initval) : Window(x, y, width, 3)
{
    fVal = initval;
    fLabelWidth = 0;
    int s = strlen(label);
    if((s > 0) && (s < (width-5)))
            SetLabel(s, label);
    PrepareContent();
    ShowValue();
}
```

(Note how arguments are passed to the base class constructor.)

Function `SetLabel()` copies the label into the left portion of the background image. Function `SetVal()` simply changes the `fVal` data member then calls `ShowValue()`.

```
void NumberItem::SetLabel(int s, char * l)
{
    fLabelWidth = s;
```

```
    for(int i=0; i< s; i++)
           fBkgd[1][i+1] = l[i];
}
```

Function `ShowValue()` starts by clearing the area used for number display. A loop is then used to generate the sequence of characters needed, these fill in the display area starting from the right. Finally, a sign is added. (If the number is too large to fit into the available display area, a set of hash marks are displayed.)

```
void NumberItem::ShowValue()
{
    int left = 2 + fLabelWidth;
    int pos = fWidth - 1;
    long val = fVal;
    for(int i = left; i<= pos; i++)
           fCurrentImg[1][i-1] = ' ';
    if(val<0) val = -val;
    if(val == 0)
           fCurrentImg[1][pos-1] = '0';
    while(val > 0) {
           int d = val % 10;
           val = val / 10;
           char ch = d + '0';
           fCurrentImg[1][pos-1] = ch;
           pos--;
           if(pos <= left) break;
           }
    if(pos<=left)
           for(i=left; i<fWidth;i++)
                  fCurrentImg[1][i-1] = '#';
    else
    if(fVal<0)
           fCurrentImg[1][pos-1] = '-';
    ShowContent();
}
```

Class `EditText` adopts a similar approach to dealing with the label, it is not shown if it would use too large a part of the window's width. The contents of the buffer have to be cleared as part of the work of the constructor (it is sufficient just to put a null character in the first element of the buffer array).

```
EditText::EditText(int x, int y, int width, char *label,
    short size) : Window(x, y, width, 3)
{
    fSize = size;
    fLabelWidth = 0;
    int s = strlen(label);
    if((s > 0) && (s < (width-8)))
           SetLabel(s, label);
    PrepareContent();
    fBuf[0] = '\0';
    ShowValue();
}
```

The `SetLabel()` function is essentially the same as that of class `NumberItem`. The `SetVal()` function loads the buffer with the given string (taking care not to overfill the array).

```
void EditText::SetVal(char* val)
{
    int n = strlen(val);
    if(n>254) n = 254;
    strncpy(fBuf,val,n);
    fBuf[n] = '\0';
    ShowValue();
}
```

The `ShowValue()` function displays the contents of the buffer, or at least that portion of the buffer that fits into the window width.

```
void EditText::ShowValue()
{
    int left = 4 + fLabelWidth;
    int i,j;
    for(i=left; i<fWidth;i++)
                  fCurrentImg[1][i-1] = ' ';
    for(i=left,j=0; i<fWidth; i++, j++) {
           char ch = fBuf[j];
           if(ch == '\0') break;
           fCurrentImg[1][i-1] = ch;
           }
    ShowContent();
}
```

Function `GetInput()` positions the cursor at the start of the data entry field then loops accepting input characters (obtained via the `WindowRep` object). The loop terminates when the required number of characters has been obtained, or when a character like a space or tab is entered.

```
char EditText::GetInput()
{
    int left = 4 + fLabelWidth;
    fEntry = 0;
    ShowValue();
    WindowRep::Instance()->MoveCursor(fX+left, fY+2);
    char ch = WindowRep::Instance()->GetChar();
    while(isalnum(ch)) {
           fBuf[fEntry] = ch;
           fEntry++;
           if(fEntry == fSize) {
                  ch = '\0';
                  break;
                  }
           ch = WindowRep::Instance()->GetChar();
           }
    fBuf[fEntry] = '\0';
    return ch;
}
```

The function does not prevent entry of long strings from overwriting parts of the screen outside of the supposed window area. You could have a more sophisticated implementation that "shifted existing text leftwards" so that display showed only the last few characters entered and text never went beyond the right margin.

29.3.2 class Dungeon

The constructor and destructor for class `Dungeon` are limited. The constructor will simply involve initializing pointer data members to `NULL`, while the destructor should delete "owned" objects like the main display window.

The `Load()` function will open the file, then use the auxiliary LoadMap() and `PopulateDungeon()` functions to read the data.

```
void Dungeon::Load(const char filename[])
{
    ifstream in(filename, ios::in | ios::nocreate);
    if(!in.good()) {
           cout << "File does not exist. Quitting." << endl;
           exit(1);
           }
    LoadMap(in);
    PopulateDungeon(in);
    in.close();
}
```

The `LoadMap()` function essentially reads "lines" of input. It will have to discard any characters that don't fit so will be making calls to `ignore()`. The argument `END_OF_LINE_CHAR` would normally be '`\n`' but some editors use '`\r`'.

```
const int END_OF_LINE_CHAR = '\r';

void Dungeon::LoadMap(ifstream& in)
{
    in >> fWidth >> fHeight;
    in.ignore(100, END_OF_LINE_CHAR);
    for(int row = 1; row <= fHeight; row++ ) {
           char ch;
           for(int col = 1; col <= fWidth; col++) {
                  in.get(ch);
                  if((row<=MAXHEIGHT) && (col <= MAXWIDTH))
                         fDRep[row-1][col-1] = ch;
                  }
           in.ignore(100, END_OF_LINE_CHAR);
           }

    if(!in.good()) {
           cout << "Sorry, problems reading that file.  "
                         "Quitting." << endl;
           exit(1);
           }
```

```
    cout << "Dungeon map read OK" << endl;

    if((fWidth > MAXWIDTH) || (fHeight > MAXHEIGHT)) {
            cout << "Map too large for window, only using "
                          "part of map." << endl;
            fWidth = (fWidth < MAXWIDTH) ? fWidth : MAXWIDTH;
            fHeight = (fHeight < MAXHEIGHT) ?
                          fHeight : MAXHEIGHT;
            }

}
```

The `DungeonItem` objects can appear in any order in the input file, but each starts with a character symbol followed by some integer data. The `PopulateDungeon()` function can use the character symbol to control a `switch()` statement in which objects of appropriate kinds are created and added to lists.

```
void Dungeon::PopulateDungeon(ifstream& in)
{
    char ch;
    Monster *m;
    in >> ch;
    while(ch != 'q') {
            switch(ch) {
case 'h':
            if(fPlayer != NULL) {
                   cout << "Limit of one player "
                          "violated." << endl;
                   exit(1);
            }
            else {
                   fPlayer = new Player(this);
                   fPlayer->Read(in);
                   }
            break;
case 'w':
            m = new Wanderer(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case 'g':
            m = new Ghost(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case 'p':
            m = new Patrol(this);
            m->Read(in);
            fInhabitants.Append(m);
            break;
case '*':
case '=':
case '$':
            Collectable *prop = new Collectable(this, ch);
            prop->Read(in);
            fProps.Append(prop);
```

Create Player object

Create different specialized Monster objects

Create Collectable items

```
                break;
        default:
                cout << "Unrecognizable data in input file."
                                << endl;
                cout << "Symbol " << ch << endl;
                exit(1);
                }
                in >> ch;
        }
        if(fPlayer == NULL) {
                cout << "No player!  No Game!" << endl;
                exit(1);
                }
        if(fProps.Length() == 0) {
                cout << "No items to collect!  No Game!" << endl;
                exit(1);
                }
        cout << "Dungeon population read" << endl;
}
```

The function verifies the requirements for exactly one `Player` object and at least one `Collectable` item.

The `Run()` function starts by creating the main map window and arranging for all objects to be drawn. The main `while()` loop shows the `Collectable` items, gets the `Player` move, then lets the `Monsters` have their turn.

```
int Dungeon::Run()
{
        CreateWindow();
        int n = fInhabitants.Length();
        for(int i=1; i <= n; i++) {
                Monster *m = (Monster*) fInhabitants.Nth(i);
                m->Draw();
                }
        fPlayer->Draw();
        fPlayer->ShowStatus();
        WindowRep::Instance()->Delay(1);


        while(fPlayer->Alive()) {
                for(int j=1; j <= fProps.Length(); j++) {
                        Collectable *pi =
                                (Collectable*) fProps.Nth(j);
                        pi->Draw();
                        }

                fPlayer->Run();
                if(fProps.Length() == 0)
                        break;

                int n = fInhabitants.Length();
                for(i=1; i<= n; i++) {
                        Monster *m = (Monster*)
                                fInhabitants.Nth(i);
                        m->Run();
```

```
                  }
            }
      return fPlayer->Alive();
}
```

(Note the need for type casts when getting members of the collections; the function `DynamicArray::Nth()` returns a void* pointer.)

The `CreateWindow()` function creates a `Window` object and sets its background from the map.

```
void Dungeon::CreateWindow()
{
    fDWindow = new Window(1, 1, fWidth, fHeight);
    for(int row = 1; row <= fHeight; row++)
           for(int col = 1; col <= fWidth; col++)
                  fDWindow->SetBkgd(col, row,
                         fDRep[row-1][col-1]);
    fDWindow->PrepareContent();
    fDWindow->ShowAll();
}
```

Class `Dungeon` has several trivial access functions:

```
int Dungeon::Accessible(Pt p) const
{
    return (' ' == fDRep[p.Y()-1][p.X()-1]);
}

Window *Dungeon::Display() { return fDWindow; }
Player *Dungeon::Human() { return fPlayer; }

int Dungeon::ValidPoint(Pt p) const
{
    int x = p.X();
    int y = p.Y();
    // check x range
    if((x <= 1) || (x >= fWidth)) return 0;
    // check y range
    if((y <= 1) || (y >= fHeight)) return 0;
    // and accessibility
    return Accessible(p);
}
```

There are similar pairs of functions `M_at_Pt()` and `PI_at_Pt()`, and `RemoveM()` and `RemoveProp()` that work with the `fInhabitants` list of `Monsters` and the `fProps` list of `Collectables`. Examples of the implementations are

```
Collectable *Dungeon::PI_at_Pt(Pt p)
{
    int n = fProps.Length();
    for(int i=1; i<= n; i++) {
           Collectable *pi = (Collectable*) fProps.Nth(i);
           Pt w = pi->Where();
           if(w.Equals(p)) return pi;
```

```
            }
    return NULL;
}

void Dungeon::RemoveM(Monster *m)
{
    fInhabitants.Remove(m);
    m->Erase();
    delete m;
}
```

The `ClearLineOfSight()` function checks the coordinates of the `Pt` arguments to determine which of the various specialized auxiliary functions should be called:

```
int Dungeon::ClearLineOfSight(Pt p1, Pt p2, int max, Pt path[])
{
    if(p1.Equals(p2)) return 0;
    if(!ValidPoint(p1)) return 0;
    if(!ValidPoint(p2)) return 0;

    if(p1.Y() == p2.Y())
            return ClearRow(p1, p2, max, path);
    else
    if(p1.X() == p2.X())
            return ClearColumn(p1, p2, max, path);

    int dx = p1.X() - p2.X();
    int dy = p1.Y() - p2.Y();

    if(abs(dx) >= abs(dy))
            return ClearSemiHorizontal(p1, p2, max, path);
    else
            return ClearSemiVertical(p1, p2, max, path);
}
```

The explanation of the algorithm given in the previous section dealt with cases involving rows or oblique lines that were more or less horizontal. The implementations given here illustrate the cases where the line is vertical or close to vertical.

```
int Dungeon::ClearColumn(Pt p1, Pt p2, int max, Pt path[])
{
    int delta = (p1.Y() < p2.Y()) ? 1 : -1;
    int x = p1.X();
    int y = p1.Y();
    for(int i = 0; i < max; i++) {
            y += delta;
            Pt p(x,y);
            if(!Accessible(p)) return 0;
            path[i] = p;
            if(p.Equals(p2)) return 1;
            }
    return 0;
}
```

```
int Dungeon::ClearSemiVertical(Pt p1, Pt p2, int max,
            Pt path[])
{
    int ychange = p2.Y() - p1.Y();
    if(abs(ychange) > max) return 0;
    int xchange = p2.X() - p1.X();

    int deltax = (xchange > 0) ? 1 : -1;
    int deltay = (ychange > 0) ? 1 : -1;

    float slope = ((float)xchange)/((float)ychange);
    float error = slope*deltay;

    int x = p1.X();
    int y = p1.Y();
    for(int i=0;i<abs(ychange);i++) {
            if(error*deltax>0.5) {
                          x += deltax;
                          error -= deltax;
                          }
            error += slope*deltay;
            y += deltay;
            Pt p(x, y);
            if(!Accessible(p)) return 0;
            path[i] = p;
            if(p.Equals(p2)) return 1;
            }
    return 0;
}
```

29.3.3 DungeonItems

DungeonItem

Class `DungeonItem` implements a few basic behaviours shared by all variants. Its constructor sets the symbol used to represent the item and sets the link to the `Dungeon` object. The body of the destructor is empty as there are no separate resources defined in the `DungeonItem` class.

```
DungeonItem::DungeonItem(Dungeon *d, char sym)
{
    fSym = sym;
    fD = d;
}

DungeonItem::~DungeonItem() { }
```

The `Erase()` and `Draw()` functions operate on the Dungeon object's main map Window. The call `fd->Display()` returns a `Window*` pointer. The `Window` referenced by this pointer is asked to perform the required operation.

```
void DungeonItem::Erase()
{
```

```
    fD->Display()->Clear(fPos.X(), fPos.Y());
}

void DungeonItem::Draw()
{
    fD->Display()->Set( fPos.X(), fPos.Y(), fSym);
}
```

All `DungeonItem` objects must read their coordinates, and the data given as input must be checked. These operations are defined in `DungeonItem::Read()`.

```
void DungeonItem::Read(ifstream& in)
{
    int x, y;
    in >> x >> y;
    if(!in.good()) {
           cout << "Problems reading coordinate data" << endl;
           exit(1);
           }
    if(!fD->ValidPoint(Pt(x,y))) {
           cout <<  "Invalid coords, out of range or"
                          "already occupied" << endl;
           cout << "(" << x << ", " << y << ")" << endl;
           exit(1);
           }
    fPos.SetPt(x,y);
}
```

Collectable

The constructor for class `Collectable` passes the `Dungeon*` pointer and `char` arguments to the `DungeonItem` constructor:

```
Collectable::Collectable(Dungeon* d, char sym) :
             DungeonItem(d, sym)
{
    fHval = fWval = fMval = 0;
}
```

Class `Collectable`'s access functions (`Wlth()` etc) simply return the values of the required data members. Its `Read()` function extends `DungeonItem::Read()`. Note the call to `DungeonItem::Read()` at the start; this gets the coordinate data. Then the extra integer parameters can be input.

Invoke inherited Read function

```
void Collectable::Read(ifstream& in)
{
    DungeonItem::Read(in);
    in >> fHval >> fWval >> fMval;
    if(!in.good()) {
           cout << "Problem reading a property" << endl;
           exit(1);
           }
}
```

ActiveItem

The constructor for class `ActiveItem` again just initializes some data members to zero after passing the given arguments to the `DungeonItem` constructor. Function `ActiveItem::Read()` is similar to `Collectable::Read()` in that it invokes the `DungeonItem::Read()` function then reads the extra data values (`fHealth` and `fStrength`).

There are a couple of trivial functions (`GetHit() { fHealth -= damage; };` and `Alive() { return fHealth > 0; }`). The `Move()` operation involves calls to the (inherited) `Erase()` and `Draw()` functions. Function `Step()` works out the x, y offset (+1, 0, or -1) coordinates of a chosen neighboring `Pt`.

```
void ActiveItem::Move(const Pt& newpoint)
{
    Erase();
    fPos.SetPt(newpoint);
    Draw();
}

Pt ActiveItem::Step(int dir)
{
    Pt p;
    switch(dir) {
case 1: p.SetPt(-1,1);    break;
case 2: p.SetPt(0,1);     break;
case 3: p.SetPt(1,1);     break;
case 4: p.SetPt(-1,0);    break;
case 6: p.SetPt(1,0);     break;
case 7: p.SetPt(-1,-1);   break;
case 8: p.SetPt(0,-1);    break;
case 9: p.SetPt(1,-1);    break;
    }
    return p;
}
```

Player

The constructor for class `Player` passes its arguments to its parents constructor and then sets its data members to 0 (`NULL` for the pointer members). The `Read()` function is similar to `Collectable::Read()`; it invokes the inherited `DungeonItem::Read()` and then gets the extra "manna" parameter.

The first call to `ShowStatus()` creates the `NumberItem` and `EditText` windows and arranges for their display. Subsequent calls update the contents of the `NumberItem` windows if there have been changes (the call to `SetVal()` results in execution of the `NumberItem` object's `ShowContents()` function so resulting in changes to the display).

```
void Player::ShowStatus()
{
    if(fWinH == NULL) {
```

```
        fWinH = new NumberItem(2, 20, 20, "Health", fHealth);
        fWinM = new NumberItem(30,20, 20, "Manna ", fManna);
        fWinW = new NumberItem(58,20, 20, "Wealth", fWealth);
        fWinE = new EditText(2, 22, 20, "Direction", 1);
        fWinH->ShowAll();
        fWinM->ShowAll();
        fWinW->ShowAll();
        fWinE->ShowAll();
        }
    else {
        if(fHealth != fWinH->GetVal()) fWinH->SetVal(fHealth);
        if(fManna != fWinM->GetVal()) fWinM->SetVal(fManna);
        if(fWealth != fWinW->GetVal()) fWinW->SetVal(fWealth);
        }
}
```

The `Run()` function involves getting and performing a command followed by update of state and display.

```
void Player::Run()
{
    char ch = GetUserCommand();
    if(isdigit(ch)) PerformMovementCommand(ch);
    else PerformMagicCommand(ch);
    UpdateState();
    ShowStatus();
}

void Player::UpdateState()
{
    fMoveCount++;
    if(0 == (fMoveCount % 3)) fHealth++;
    if(0 == (fMoveCount % 7)) fManna++;
}
```

The function `PeformMovementCommand()` first identifies the neighboring point. There is then an interaction with the `Dungeon` object to determine whether there is a `Collectable` at that point (if so, it gets taken). A similar interaction determines whether there is a `Monster` (if so, it gets attacked, after which a return is made from this function). If the neighboring point is not occupied by a `Monster`, the `Player` object moves to that location.

```
void Player::PerformMovementCommand(char ch)
{
    int x = fPos.X();
    int y = fPos.Y();
    Pt p = Step(ch - '0');
    int newx = x + p.X();
    int newy = y + p.Y();

    Collectable *pi = fD->PI_at_Pt(Pt(newx, newy));
    if(pi != NULL)
           Take(pi);
    Monster *m = fD->M_at_Pt(Pt(newx, newy));
```

```
        if(m != NULL) {
                Attack(m);
                return;
                }
        TryMove(x + p.X(), y + p.Y());
    }
```

The auxiliary functions, `Take()`, `Attack()`, and `TryMove()` are all simple. Function `Take()` updates the `Player` objects health and related attributes with data values from the `Collectable` item, and then arranges for the `Dungeon` to dispose of that item. Function `Attack()` reduces the `Monster` object's health (via a call to its `GetHit()` function) and, if appropriate, arranges for the `Dungeon` object to dispose of the `Monster`. Function `TryMove()` validates and then performs the appropriate movement.

The function `GetUserCommand()` arranges for the `EditText` window to input some text and then inspects the first character of the text entered.

```
    char Player::GetUserCommand()
    {
        fWinE->GetInput();
        char *str = fWinE->GetVal();
        return *str;
    }
```

The function `PerformMagicCommand()` identifies the axis for the magic bolt. There is then a loop in which damage is inflicted (at a reducing rate) on any `Monster` objects found along a sequence of points in the given direction:

```
    void Player::PerformMagicCommand(char ch)
    {
        int dx, dy;
        switch (ch) {
    case 'q': dx = -1; dy = -1; break;
    case 'w': dx = 0; dy = -1; break;
    case 'e': dx = 1; dy = -1; break;
    case 'a': dx = -1; dy = 0; break;
    case 'd': dx = 1; dy = 0; break;
    case 'z': dx = -1; dy = 1; break;
    case 'x': dx = 0; dy = 1; break;
    case 'c': dx = 1; dy = 1; break;
    default:
                return;
        }
        int x = fPos.X();
        int y = fPos.Y();

        int power = 8;
        fManna -= power;
        if(fManna < 0) {
                fHealth += 2*fManna;
                fManna = 0;
                }
        while(power > 0) {
                x += dx;
```

```
                y += dy;
                if(!fD->ValidPoint(Pt(x,y))) return;
                Monster* m = fD->M_at_Pt(Pt(x,y));
                if(m != NULL) {
                       m->GetHit(power);
                       if(!m->Alive())
                              fD->RemoveM(m);
                       }
                power /= 2;
                }
    }
```

Monster

The constructor and destructor functions of class `Monster` both have empty bodies for there is no work to be done; the constructor passes its arguments back to the constructor of its parent class (`ActiveItem`):

```
Monster::Monster(Dungeon *d, char sym) : ActiveItem(d, sym)
{
}
```

Function `Monster::Run()` was defined earlier. The default implementations of the auxiliary functions are:

```
int Monster::CanAttack()
{
    Player *p = fD->Human();
    Pt target = p->Where();
    return fPos.Adjacent(target);
}

void Monster::Attack()
{
    Player *p = fD->Human();
    p->GetHit(fStrength);
}

int Monster::CanDetect() { return 0; }

void Monster::Advance() { }
```

Ghost

The `Ghost::CanDetect()` function uses the `Pt::Distance()` member function to determine the distance to the Player (this function just takes the normal Euclidean distance between two points, rounded up to the next integral value).

```
int Ghost::CanDetect()
{
    Player *p = fD->Human();
```

```
    int range = fPos.Distance(p->Where());
    return (range < 7);
}
```

The `Advance()` function determines the change in x, y coords that will bring the `Ghost` closer to the `Player`.

```
void Ghost::Advance()
{
    Player *p = fD->Human();
    Pt p1 = p->Where();
    int dx, dy;
    dx = dy = 0;
    if(p1.X() > fPos.X()) dx = 1;
    else
    if(p1.X() < fPos.X()) dx = -1;
    if(p1.Y() > fPos.Y()) dy = 1;
    else
    if(p1.Y() < fPos.Y()) dy = -1;

    Move(Pt(fPos.X() + dx, fPos.Y() + dy));
}
```

Wanderer

The `Wanderer::CanDetect()` function uses the `Dungeon::ClearLineOfSight()` member function to determine whether the `Player` object is visible. This function call also fills in the array `fPath` with the points that will have to be crossed.

```
int Wanderer::CanDetect()
{
    Player *p = fD->Human();
    return
      fD->ClearLineOfSight(fPos, p->Where(), 10, fPath);
}
```

The `Advance()` function moves one step along the path:

```
void Wanderer::Advance()
{
    Move(fPath[0]);
}
```

The `NormalMove()` function tries moving in the same direction as before. Directions are held by storing the delta-x and delta-y values in `fLastX` and `fLastY` data members (initialized to zero in the constructor). If movement in that general direction is blocked, a new direction is picked randomly.

```
void Wanderer::NormalMove()
{
    int x = fPos.X();
    int y = fPos.Y();
```

Movement in same direction

Movement in similar direction

Pick new direction at random

```
        // Try to keep going in much the same direction as last time
        if((fLastX != 0) || (fLastY != 0)) {
                int newx = x + fLastX;
                int newy = y + fLastY;
                if(fD->Accessible(Pt(newx,newy))) {
                        Move(Pt(newx,newy));
                        return;
                        }
                else
                if(fD->Accessible(Pt(newx,y))) {
                        Move(Pt(newx,y)); fLastY = 0;
                        return;
                        }
                else
                if(fD->Accessible(Pt(x,newy))) {
                        Move(Pt(x,newy)); fLastX= 0;
                        return; }
                }
        int dir = rand();
        dir = dir % 9;
        dir++;
        Pt p = Step(dir);
        x += p.X();
        y += p.Y();
        if(fD->Accessible(Pt(x,y))) {
                fLastX = p.X();
                fLastY = p.Y();
                Move(Pt(x,y));
                }
}
```

Patrol

The patrol route has to be read, consequently the inherited `Read()` function must be extended. There are several possible errors in route definitions, so `Patrol::Read()` involves many checks:

```
void Patrol::Read(ifstream& in)
{
        Monster::Read(in);
        fRoute[0] = fPos;
        fNdx = 0;
        fDelta = 1;
        in >> fRouteLen;
        for(int i=1; i<= fRouteLen; i++) {
                int x, y;
                in >> x >> y;
                Pt p(x, y);
                if(!fD->ValidPoint(p)) {
                        cout << "Bad data in patrol route" << endl;
                        cout << "(" << x << ", " << y << ")" <<
endl;
                        exit(1);
                        }
```

```
            if(!p.Adjacent(fRoute[i-1])) {
                   cout << "Non adjacent points in patrol"
                                  "route" << endl;
                   cout << "(" << x << ", " << y << ")" << endl;
                   exit(1);
                   }
            fRoute[i] = p;
            }
    if(!in.good()) {
            cout << "Problems reading patrol route" << endl;
            exit(1);
            }
}
```

The `NormalMove()` function causes a `Patrol` object to move up or down its route:

```
void Patrol::NormalMove()
{
    if((fNdx == 0) && (fDelta == -1)) {
            fDelta = 1;
            return;
            }
    if((fNdx == fRouteLen) && (fDelta == 1)) {
            fDelta = -1;
            return;
            }
    fNdx += fDelta;
    Move(fRoute[fNdx]);
}
```

Reverse direction at start

Reverse direction at end

Move one step along route

The `CanDetect()` function is identical to `Wanderer::CanDect()`. However, instead of advancing one step along the path to the `Player`, a `Patrol` fires a projectile that moves along the complete path. When the projectile hits, it causes a small amount of damage:

```
void Patrol::Advance()
{
    Player *p = fD->Human();
    Pt target = p->Where();
    Pt arrow = fPath[0];
    int i = 1;
    while(!arrow.Equals(target)) {
            fD->Display()->Set( arrow.X(), arrow.Y(), ':');
            WindowRep::Instance()->Delay(1);
            fD->Display()->Clear( arrow.X(), arrow.Y());
            arrow = fPath[i];
            i++;
            }
    p->GetHit(2);
}
```

EXERCISES

1 Complete and run the dungeon game program.

2 This one is only for users of Borland's system.

Why should the monsters wait while the user thinks? If they know what they want to do, they should be able to continue!

The current program requires user input in each cycle of the game. If there is no input, the program stops and waits. The game is much more interesting if this wait is limited. If the user doesn't type any command within a second or so, the monsters should get their chance to run anyway.

This is not too hard to arrange.

First, the main while() loop in Dungeon::Run() should have a call WindowRep:: Instance()->Delay(1). This results in a 1 second pause in each cycle.

The Player::Run() function only gets called if there have been some keystrokes. If there are no keystrokes waiting to be processed, the Dungeon::Run() function skips to the loop that lets each monster have a chance to run.

All that is required is a system function, in the "console" library package, that allows a program to check whether input data are available (without "blocking" like a normal read function). The Borland conio library includes such a function.

Using the on-line help system in the Borland environment, and other printed documentation, find how to check for input. Use this function in a reorganized version of the dungeon program.

(You can achieve the same result in the Symantec system but only by utilising specialized system calls to the "Toolbox" component of the Macintosh operating system. It is all a little obscure and clumsy.)

3 Add multiple levels to the dungeon.

(There are various ways that this might be done. The easiest is probably to define a new class DungeonLevel . The Dungeon object owns the main window, the Player, and a list of DungeonLevel objects. Each DungeonLevel object owns a map, a list of collectables, and a list of monsters. You will need some way of allowing a user to go up or down levels. When you change level, the new DungeonLevel resets the background map in the main window and arranges for all data to be redrawn.)

4 Add more challenging Monsters and "traps".

(Use your own imagination.)