3.5 APPLICATION OF QUEUES: SIMULATION

3.5.1 Introduction

simulation

Simulation is the use of one system to imitate the behavior of another system. Simulations are often used when it would be too expensive or dangerous to experiment with the real system. There are physical simulations, such as wind tunnels used to experiment with designs for car bodies and flight simulators used to train airline pilots. Mathematical simulations are systems of equations used to describe some system, and computer simulations use the steps of a program to imitate the behavior of the system under study.

computer simulation

In a computer simulation, the objects being studied are usually represented as data, often as data structures given by classes whose members describe the properties of the objects. Actions being studied are represented as methods of the classes, and the rules describing these actions are translated into computer algorithms. By changing the values of the data or by modifying these algorithms, we can observe the changes in the computer simulation, and then we can draw worthwhile inferences concerning the behavior of the actual system.

While one object in a system is involved in some action, other objects and actions will often need to be kept waiting. Hence queues are important data structures for use in computer simulations. We shall study one of the most common and useful kinds of computer simulations, one that concentrates on queues as its basic data structure. These simulations imitate the behavior of systems (often, in fact, called ***queueing systems***) in which there are queues of objects waiting to be served by various processes.

3.5.2 Simulation of an Airport

As a specific example, let us consider a small but busy airport with only one runway (see Figure 3.5). In each unit of time, one plane can land or one plane can take off, but not both. Planes arrive ready to land or to take off at random times, so at any given moment of time, the runway may be idle or a plane may be landing or taking off, and there may be several planes waiting either to land or take off.

class Plane

In simulating the airport, it will be useful to create a class Plane whose objects represent individual planes. This class will definitely need an initialization method and methods to represent takeoff and landing. Moreover, when we write the main program for the simulation, the need for other Plane methods will become apparent. We will also use a class Runway to hold information about the state and operation of the runway. This class will maintain members representing queues of planes waiting to land and take off.

class Random

We shall need one other class in our simulation, a class Random to encapsulate the random nature of plane arrivals and departures from the runway. We shall discuss this class in more detail in Section 3.5.3. In our main program, we use a

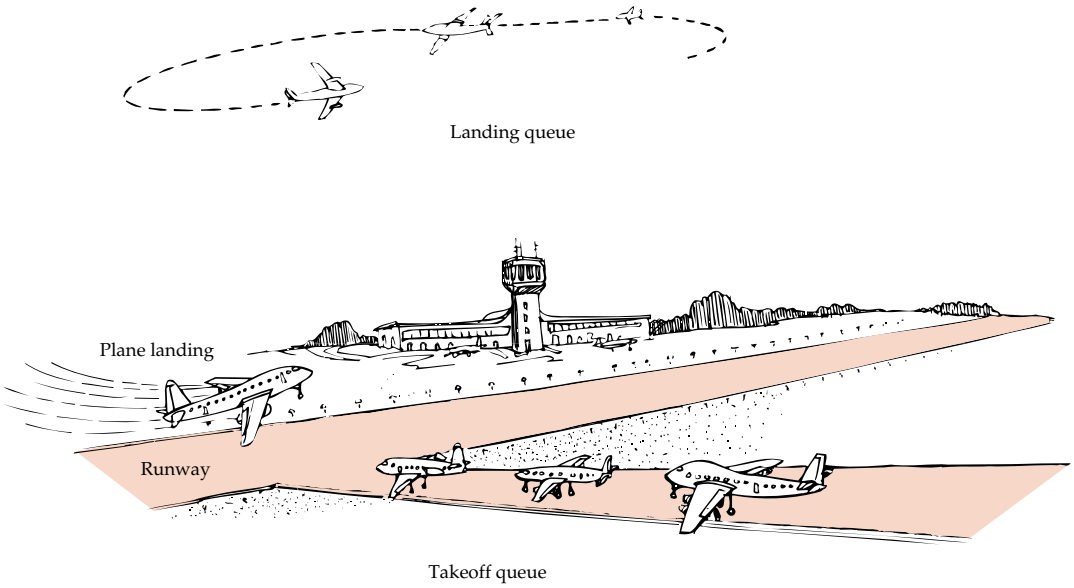

Figure 3.5. **An airport**

67

single method, called poisson, from the **class** Random. This method uses a floating-point parameter (representing an average outcome) and it returns an integer value. Although the returned value is random, it has the property that over the course of many repeated method calls, the average of the returned values will match our specified parameter.

In our simulation, we shall be especially concerned with the amounts of time that planes need to wait in queues before taking off or landing. Therefore, the measurement of time will be of utmost importance to our program. We shall divide the time period of our simulation into units in such a way that just one plane can use the runway, either to land or take off, in any given unit of time.

The precise details of how we handle the landing and takeoff queues will be dealt with when we program the Runway class. Similarly, the precise methods describing the operation of a Plane are not needed by our main program.

68

```
int main()                          //  Airport simulation program
/* Pre:  The user must supply the number of time intervals the simulation is to run,
         the expected number of planes arriving, the expected number of planes
         departing per time interval, and the maximum allowed size for runway
         queues.
   Post: The program performs a random simulation of the airport, showing the
         status of the runway at each time interval, and prints out a summary of
         airport operation at the conclusion.
   Uses: Classes Runway, Plane, Random and functions run_idle, initialize. */
```

```
{
  int end_time;                    // time to run simulation
  int queue_limit;                 // size of Runway queues
  int flight_number = 0;
  double arrival_rate, departure_rate;
  initialize(end_time, queue_limit, arrival_rate, departure_rate);
  Random variable;
  Runway small_airport(queue_limit);
  for (int current_time = 0; current_time < end_time; current_time++) {
                                   // loop over time intervals
    int number_arrivals = variable.poisson(arrival_rate);
                                   // current arrival requests
    for (int i = 0; i < number_arrivals; i++) {
      Plane current_plane(flight_number++, current_time, arriving);
      if (small_airport.can_land(current_plane) != success)
        current_plane.refuse( );
    }
    int number_departures = variable.poisson(departure_rate);
                                   // current departure requests
    for (int j = 0; j < number_departures; j++) {
      Plane current_plane(flight_number++, current_time, departing);
      if (small_airport.can_depart(current_plane) != success)
        current_plane.refuse( );
    }

    Plane moving_plane;
    switch (small_airport.activity(current_time, moving_plane)) {
     // Let at most one Plane onto the Runway at current_time.
    case land:
      moving_plane.land(current_time);
      break;
    case takeoff:
      moving_plane.fly(current_time);
      break;
    case idle:
      run_idle(current_time);
    }
  }
  small_airport.shut_down(end_time);
}
```

In this program, we begin with a call to the function initialize that prints instructions to the user and gathers information about how long the user wishes the simulation to run and how busy the airport is to be. We then enter a **for** loop, in which current_time ranges from 0 to the user specified value end_time. In each time unit, we process random numbers of arriving and departing planes; these planes are declared and initialized as the objects called current_plane. In each cycle, we also allow one moving plane to use the runway. If there is no plane to use the runway,

we apply the function run_idle. Note that if our **class** Runway is unable to add an incoming flight to the landing Queue (presumably because the Queue is full), we apply a method called refuse to direct the Plane to another airport. Similarly, we sometimes have to refuse a Plane permission to take off.

3.5.3 Random Numbers

A key step in our simulation is to decide, at each time unit, how many new planes become ready to land or take off. Although there are many ways in which these decisions can be made, one of the most interesting and useful is to make a random decision. When the program is run repeatedly with random decisions, the results will differ from run to run, and with sufficient experimentation, the simulation may display a range of behavior not unlike that of the actual system being studied. The Random method poisson in the preceding main program returns a random number of planes arriving ready to land or ready to take off in a particular time unit.

pseudorandom number

Appendix B studies numbers, called ***pseudorandom***, for use in computer programs. Several different kinds of pseudorandom numbers are useful for different applications. For the airport simulation, we need one of the more sophisticated kinds, called *Poisson* random numbers.

To introduce the idea, let us note that saying that an average family has 2.6 children does not mean that each family has 2 children and 0.6 of a third. Instead, it means that, averaged over many families, the mean number of children is 2.6. Hence, for five families with 4, 1, 0, 3, 5 children the mean number is 2.6. Similarly, if the number of planes arriving to land in ten time units is 2, 0, 0, 1, 4, 1, 0, 0, 0, 1, then the mean number of planes arriving in one unit is 0.9.

expected value, Poisson distribution

Let us now start with a fixed number called the ***expected value*** ν of the random numbers. Then to say that a sequence of nonnegative integers satisfies a ***Poisson distribution*** with expected value ν implies that, over long subsequences, the mean value of the integers in the sequence approaches ν. Appendix B describes a C++ class that generates random integers according to a Poisson distribution with a given expected value, and this is just what we need for the airport simulation.

3.5.4 The Runway Class Specification

rules

The Runway class needs to maintain two queues of planes, which we shall call landing and takeoff, to hold waiting planes. It is better to keep a plane waiting on the ground than in the air, so a small airport allows a plane to take off only if there are no planes waiting to land. Hence, our Runway method activity, which controls access to the Runway, will first service the head of the Queue of planes waiting to land, and only if the landing Queue is empty will it allow a Plane to take off.

One aim of our simulation is to gather data about likely airport use. It is natural to use the **class** Runway itself to keep statistics such the number of planes processed, the average time spent waiting, and the number of planes (if any) refused service. These details are reflected in the various data members of the following Runway class definition.

```
enum Runway_activity {idle, land, takeoff};
```

Runway *definition*

```
class Runway {
public:
  Runway(int limit);
  Error_code can_land(const Plane &current);
  Error_code can_depart(const Plane &current);
  Runway_activity activity(int time, Plane &moving);
  void shut_down(int time) const;
private:
  Extended_queue landing;
  Extended_queue takeoff;
  int queue_limit;
  int num_land_requests;          //  number of planes asking to land
  int num_takeoff_requests;       //  number of planes asking to take off
  int num_landings;               //  number of planes that have landed
  int num_takeoffs;               //  number of planes that have taken off
  int num_land_accepted;          //  number of planes queued to land
  int num_takeoff_accepted;       //  number of planes queued to take off
  int num_land_refused;           //  number of landing planes refused
  int num_takeoff_refused;        //  number of departing planes refused
  int land_wait;                  //  total time of planes waiting to land
  int takeoff_wait;               //  total time of planes waiting to take off
  int idle_time;                  //  total time runway is idle
};
```

Note that the **class** Runway has two queues among its members. The implementation reflects the has-a relationships in the statement that a runway has a landing queue and has a takeoff queue.

3.5.5 The Plane Class Specification

The **class** Plane needs to maintain data about particular Plane objects. This data must include a flight number, a time of arrival at the airport system, and a Plane status as either arriving or departing. Since we do not wish a client to be able to change this information, we shall keep it in private data members. When we declare a Plane object in the main program, we shall wish to initialize these three pieces of information as the object is constructed. Hence we need a Plane class constructor that has three parameters. Other times, however, we shall wish to construct a Plane object without initializing this information, because either its values are irrelevant or will otherwise be determined. Hence we really need two constructors for the Plane class, one with three parameters and one with none.

multiple versions of functions

The C++ language provides exactly the feature we need; it allows us to use the same identifier to name as many different functions as we like, even within a single block of code, so long as no two of these functions have identically typed parameter lists. When the function is invoked, the C++ compiler can figure out which version of the function to use, by looking at the number of actual parameters and their types. It simply determines which set of formal parameters match the actual parameters in number and types.

function overloading

When we use a single name for several different functions, we say that the name is ***overloaded***. Inside the scope of the **class** Plane, we are able to overload the two plane constructors, because the first uses an empty parameter list, whereas the second uses a parameter list of three integer variables.

From now on, class specifications will often contain two constructors, one with parameters for initializing data members, and one without parameters.

Finally, the Plane class must contain the methods refuse, land, and fly that are explicitly used by the main program. We will also need each Plane to be able to communicate its time of arrival at the airport to the **class** Runway, so a final method called started is included with this purpose in mind. We can now give the specification for the **class** Plane.

```
enum Plane_status {null, arriving, departing};
```

Plane *definition*

```
class Plane {
public:
  Plane( );
  Plane(int flt, int time, Plane_status status);
  void refuse( ) const;
  void land(int time) const;
  void fly(int time) const;
  int started( ) const;
private:
  int flt_num;
  int clock_start;
  Plane_status state;
};
```

3.5.6 Functions and Methods of the Simulation

The actions of the functions and methods for doing the steps of the simulation are generally straightforward, so we proceed to write each in turn, with comments only as needed for clarity.

1. Simulation Initialization

```
void initialize(int &end_time, int &queue_limit,
                double &arrival_rate, double &departure_rate)
/* Pre:  The user specifies the number of time units in the simulation, the maximal
         queue sizes permitted, and the expected arrival and departure rates for
         the airport.
   Post: The program prints instructions and initializes the parameters end_time,
         queue_limit, arrival_rate, and departure_rate to the specified values.
   Uses: utility function user_says_yes */
{
  cout << "This program simulates an airport with only one runway." << endl
       << "One plane can land or depart in each unit of time." << endl;
  cout << "Up to what number of planes can be waiting to land "
       << "or take off at any time? " << flush;
  cin >> queue_limit;
  cout << "How many units of time will the simulation run?" << flush;
  cin >> end_time;
  bool acceptable;
  do {
    cout << "Expected number of arrivals per unit time?" << flush;
    cin >> arrival_rate;
    cout << "Expected number of departures per unit time?" << flush;
    cin >> departure_rate;
    if (arrival_rate < 0.0 || departure_rate < 0.0)
      cerr << "These rates must be nonnegative." << endl;
    else
      acceptable = true;
    if (acceptable && arrival_rate + departure_rate > 1.0)
      cerr << "Safety Warning: This airport will become saturated. " << endl;
  } while (!acceptable);
}
```

2. Runway Initialization

```
Runway::Runway(int limit)
/* Post: The Runway data members are initialized to record no prior Runway use
         and to record the limit on queue sizes. */
{
  queue_limit = limit;
  num_land_requests = num_takeoff_requests = 0;
  num_landings = num_takeoffs = 0;
  num_land_refused = num_takeoff_refused = 0;
  num_land_accepted = num_takeoff_accepted = 0;
  land_wait = takeoff_wait = idle_time = 0;
}
```

3. Accepting a New Plane into a Runway Queue

```
Error_code Runway::can_land(const Plane &current)
/* Post: If possible, the Plane current is added to the landing Queue; otherwise,
         an Error_code of overflow is returned. The Runway statistics are updated.
   Uses: class Extended_queue. */
{
  Error_code result;
  if (landing.size( ) < queue_limit)
    result = landing.append(current);
  else
    result = fail;
  num_land_requests++;
  if (result != success)
    num_land_refused++;
  else
    num_land_accepted++;
  return result;
}

Error_code Runway::can_depart(const Plane &current)
/* Post: If possible, the Plane current is added to the takeoff Queue; otherwise, an
         Error_code of overflow is returned. The Runway statistics are updated.
   Uses: class Extended_queue. */
{
  Error_code result;
  if (takeoff.size( ) < queue_limit)
    result = takeoff.append(current);
  else
    result = fail;
  num_takeoff_requests++;
  if (result != success)
    num_takeoff_refused++;
  else
    num_takeoff_accepted++;
  return result;
}
```

4. Handling Runway Access

```
Runway_activity Runway::activity(int time, Plane &moving)
/* Post: If the landing Queue has entries, its front Plane is copied to the parameter
         moving and a result land is returned. Otherwise, if the takeoff Queue has
         entries, its front Plane is copied to the parameter moving and a result
         takeoff is returned. Otherwise, idle is returned. Runway statistics are
         updated.
   Uses: class Extended_queue. */
```

```
{
  Runway_activity in_progress;
  if (!landing.empty( )) {
    landing.retrieve(moving);
    land_wait += time - moving.started( );
    num_landings++;
    in_progress = land;
    landing.serve( );
  }
  else if (!takeoff.empty( )) {
    takeoff.retrieve(moving);
    takeoff_wait += time - moving.started( );
    num_takeoffs++;
    in_progress = takeoff;
    takeoff.serve( );
  }
  else {
    idle_time++;
    in_progress = idle;
  }
  return in_progress;
}
```

5. Plane Initialization

```
Plane::Plane(int flt, int time, Plane_status status)
/* Post: The Plane data members flt_num, clock_start, and state are set to the
         values of the parameters flt, time and status, respectively. */
{
  flt_num = flt;
  clock_start = time;
  state = status;
  cout << "Plane number " << flt << " ready to ";
  if (status == arriving)
    cout << "land." << endl;
  else
    cout << "take off." << endl;
}

Plane::Plane( )
/* Post: The Plane data members flt_num, clock_start, state are set to illegal de-
         fault values. */
{
  flt_num = -1;
  clock_start = -1;
  state = null;
}
```

null initialization

The second of these constructors performs a ***null initialization***. In many programs it is not necessary to provide such a constructor for a class. However, in C++, if we ever declare an array of objects that do have a constructor, then the objects must have an explicit default constructor. A ***default constructor*** is a constructor without parameters (or with specified defaults for all parameters). Each Runway object contains queues of planes, and each of these queues is implemented using an array of planes. Hence, in our simulation, we really do need the null initialization operation.

6. Refusing a Plane

```
void Plane::refuse( ) const
/* Post: Processes a Plane wanting to use Runway, when the Queue is full. */
{
  cout << "Plane number " << flt_num;
  if (state == arriving)
    cout << " directed to another airport" << endl;
  else
    cout << " told to try to takeoff again later" << endl;
}
```

7. Processing an Arriving Plane

```
void Plane::land(int time) const
/* Post: Processes a Plane that is landing at the specified time. */
{
  int wait = time - clock_start;
  cout  << time << ": Plane number " << flt_num << " landed after "
        << wait << " time unit" << ((wait == 1) ? "" : "s")
        << " in the takeoff queue." << endl;
}
```

In this function we have used the ternary operator ? : to append an "s" where needed to achieve output such as "1 time unit" or "2 time units".

8. Processing a Departing Plane

```
void Plane::fly(int time) const
/* Post: Process a Plane that is taking off at the specified time. */
{
  int wait = time - clock_start;
  cout  << time << ": Plane number " << flt_num << " took off after "
        << wait << " time unit" << ((wait == 1) ? "" : "s")
        << " in the takeoff queue." << endl;
}
```

9. Communicating a Plane's Arrival Data

```
int Plane :: started( ) const
/* Post: Return the time that the Plane entered the airport system. */
{
   return clock_start;
}
```

10. Marking an Idle Time Unit

```
void run_idle(int time)
/* Post: The specified time is printed with a message that the runway is idle. */
{
   cout << time << ": Runway is idle." << endl;
}
```

11. Finishing the Simulation

```
void Runway :: shut_down(int time) const
/* Post: Runway usage statistics are summarized and printed. */
{
   cout << "Simulation has concluded after " << time << " time units." << endl
        << "Total number of planes processed "
        << (num_land_requests + num_takeoff_requests) << endl
        << "Total number of planes asking to land "
        << num_land_requests << endl
        << "Total number of planes asking to take off "
        << num_takeoff_requests << endl
        << "Total number of planes accepted for landing "
        << num_land_accepted << endl
        << "Total number of planes accepted for takeoff "
        << num_takeoff_accepted << endl
        << "Total number of planes refused for landing "
        << num_land_refused << endl
        << "Total number of planes refused for takeoff "
        << num_takeoff_refused << endl
        << "Total number of planes that landed "
        << num_landings << endl
        << "Total number of planes that took off "
        << num_takeoffs << endl
        << "Total number of planes left in landing queue "
        << landing.size( ) << endl
        << "Total number of planes left in takeoff queue "
        << takeoff.size( ) << endl;
```

```
    cout << "Percentage of time runway idle "
         << 100.0 * ((float) idle_time)/((float) time) << "%" << endl;
    cout << "Average wait in landing queue "
         << ((float) land_wait)/((float) num_landings) << " time units";
    cout << endl << "Average wait in takeoff queue "
         << ((float) takeoff_wait)/((float) num_takeoffs)
         << " time units" << endl;
    cout << "Average observed rate of planes wanting to land "
         << ((float) num_land_requests)/((float) time)
         << " per time unit" << endl;
    cout << "Average observed rate of planes wanting to take off "
         << ((float) num_takeoff_requests)/((float) time)
         << " per time unit" << endl;
}
```

3.5.7 Sample Results

We conclude this section with part of the output from a sample run of the airport simulation. You should note that there are some periods when the runway is idle and others when one of the queues is completely full and in which planes must be turned away. If you run this simulation again, you will obtain different results from those given here, but, if the expected values given to the program are the same, then there will be some correspondence between the numbers given in the summaries of the two runs.

```
This program simulates an airport with only one runway.
One plane can land or depart in each unit of time.
Up to what number of planes can be waiting to land or take off at any time ? 5
How many units of time will the simulation run ? 1000
Expected number of arrivals per unit time ? .48
Expected number of departures per unit time ? .48
        Plane number 0 ready to take off.
0:      Plane 1 landed; in queue 0 units.
        Plane number 0 took off after 0 time units in the takeoff queue.
        Plane number 1 ready to take off.
1:      Plane number 1 took off after 0 time units in the takeoff queue.
        Plane number 2 ready to take off.
        Plane number 3 ready to take off.
2:      Plane number 2 took off after 0 time units in the takeoff queue.
        Plane number 4 ready to land.
        Plane number 5 ready to take off.
```

```
 3: Plane number 4 landed after 0 time units in the takeoff queue.
    Plane number 6 ready to land.
    Plane number 7 ready to land.
    Plane number 8 ready to take off.
    Plane number 9 ready to take off.
 4: Plane number 6 landed after 0 time units in the takeoff queue.
    Plane number 10 ready to land.
    Plane number 11 ready to take off.
 5: Plane number 7 landed after 1 time unit in the takeoff queue.
    Plane number 12 ready to land.
 6: Plane number 10 landed after 1 time unit in the takeoff queue.
 7: Plane number 12 landed after 1 time unit in the takeoff queue.
    Plane number 13 ready to land.
    Plane number 14 ready to take off.
takeoff queue is full
    Plane number 14 told to try to takeoff again later.
 8: Plane number 13 landed after 0 time units in the takeoff queue.
 9: Plane number 3 took off after 7 time units in the takeoff queue.
10: Plane number 5 took off after 7 time units in the takeoff queue.
11: Plane number 8 took off after 7 time units in the takeoff queue.
    Plane number 15 ready to take off.
12: Plane number 9 took off after 8 time units in the takeoff queue.
    Plane number 16 ready to land.
    Plane number 17 ready to land.
13: Plane number 16 landed after 0 time units in the takeoff queue.
    Plane number 18 ready to land.
14: Plane number 17 landed after 1 time unit in the takeoff queue.
15: Plane number 18 landed after 1 time unit in the takeoff queue.
    Plane number 19 ready to land.
    Plane number 20 ready to take off.
16: Plane number 19 landed after 0 time units in the takeoff queue.
17: Plane number 11 took off after 12 time units in the takeoff queue.
18: Plane number 15 took off after 6 time units in the takeoff queue.
19: Plane number 20 took off after 3 time units in the takeoff queue.
both queues are empty
20: Runway is idle.
```

Eventually, after many more steps of the simulation, we get a statistical summary.

summary

Simulation has concluded after 1000 time units.

Total number of planes processed	970
Total number of planes asking to land	484
Total number of planes asking to take off	486
Total number of planes accepted for landing	484
Total number of planes accepted for takeoff	423
Total number of planes refused for landing	0
Total number of planes refused for takeoff	63
Total number of planes that landed	483
Total number of planes that took off	422
Total number of planes left in landing queue	1
Total number of planes left in takeoff queue	1
Percentage of time runway idle	9.5 %
Average wait in landing queue	0.36646 time units
Average wait in takeoff queue	4.63744 time units
Average observed rate of planes wanting to land	0.484 time units
Average observed rate of planes wanting to take off	0.486 time units

Notice that the last two statistics, giving the observed rates of planes asking for landing and departure permission, do match the expected values put in at the beginning of the run (within a reasonable range): This outcome should give us some confidence that the pseudo-random number algorithm of Appendix B really does simulate an appropriate Poisson distribution.

Programming Projects 3.5

P1. Combine all the functions and methods for the airport simulation into a complete program. Experiment with several sample runs of the airport simulation, adjusting the values for the expected numbers of planes ready to land and take off. Find approximate values for these expected numbers that are as large as possible subject to the condition that it is very unlikely that a plane must be refused service. What happens to these values if the maximum size of the queues is increased or decreased?

P2. Modify the simulation to give the airport two runways, one always used for landings and one always used for takeoffs. Compare the total number of planes that can be served with the number for the one-runway airport. Does it more than double?

P3. Modify the simulation to give the airport two runways, one usually used for landings and one usually used for takeoffs. If one of the queues is empty, then both runways can be used for the other queue. Also, if the landing queue is full and another plane arrives to land, then takeoffs will be stopped and both runways used to clear the backlog of landing planes.

P4. Modify the simulation to have three runways, one always reserved for each of landing and takeoff and the third used for landings unless the landing queue is empty, in which case it can be used for takeoffs.

P5. Modify the original (one-runway) simulation so that when each plane arrives to land, it will have (as one of its data members) a (randomly generated) fuel level, measured in units of time remaining. If the plane does not have enough fuel to wait in the queue, it is allowed to land immediately. Hence the planes in the landing queue may be kept waiting additional units, and so may run out of fuel themselves. Check this out as part of the landing function, and find about how busy the airport can become before planes start to crash from running out of fuel.

P6. Write a stub to take the place of the random-number function. The stub can be used both to debug the program and to allow the user to control exactly the number of planes arriving for each queue at each time unit.

POINTERS AND PITFALLS

1. Before choosing implementations, be sure that all the data structures and their associated operations are fully specified on the abstract level.
2. In choosing between implementations, consider the necessary operations on the data structure.
3. If every object of **class** A has all the properties of an object of **class** B, implement **class** A as a derived class of B.
4. Consider the requirements of derived classes when declaring the members of a base class.
5. Implement is-a relationships between classes by using public inheritance.
6. Implement has-a relationships between classes by layering.
7. Use Poisson random variables to model random event occurrences.

REVIEW QUESTIONS

3.1 **1.** Define the term *queue*. What operations can be done on a queue?

2. How is a circular array implemented in a linear array?

3. List three different implementations of queues.

4. Explain the difference between *has-a* and *is-a* relationships between classes.

3.4 **5.** Define the term *simulation*.