Chapter 3

QUEUES

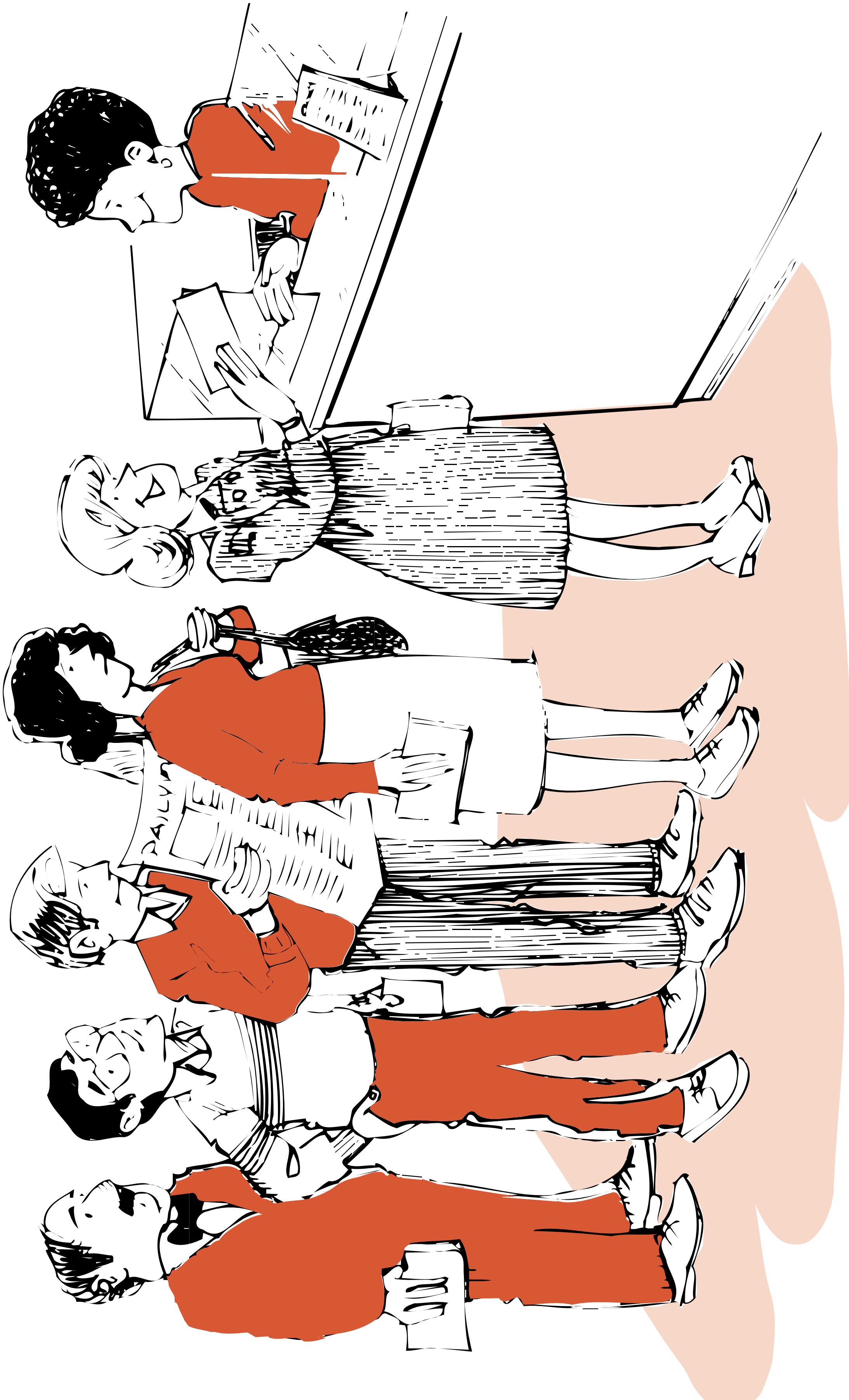
1. Specifications for Queues

2. Implementations of Queues

3. Contiguous Queues in C++

4. Demonstration and Testing

5. Application: Airport Simulation



A queue

Specifications for Queues

Queue :: Queue( );

Post: The Queue has been created and is initialized to be empty.

Error_code Queue :: append(**const** Queue_entry &x);

Post: If there is space, x is added to the Queue as its rear. Otherwise an Error_code of overflow is returned.

Error_code Queue :: serve( );

Post: If the Queue is not empty, the front of the Queue has been removed. Otherwise an Error_code of underflow is returned.

Error_code Queue :: retrieve(Queue_entry &x) **const**;

Post: If the Queue is not empty, the front of the Queue has been recorded as x. Otherwise an Error_code of underflow is returned.

bool Queue :: empty( ) **const**;

Post: Return **true** if the Queue is empty, otherwise return **false**.

Extended Queues

```
class Extended_queue: public Queue {
public:
   bool full( ) const;
   int size( ) const;
   void clear( );
   Error_code serve_and_retrieve(Queue_entry &item);
};
```

bool Extended_queue :: full( ) **const**;

Post: Return **true** if the Extended_queue is full; return **false** otherwise.

void Extended_queue :: clear( );

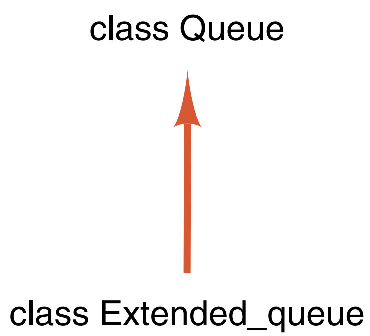
Post: All entries in the Extended_queue have been removed; it is now empty.

int Extended_queue :: size( ) **const**;

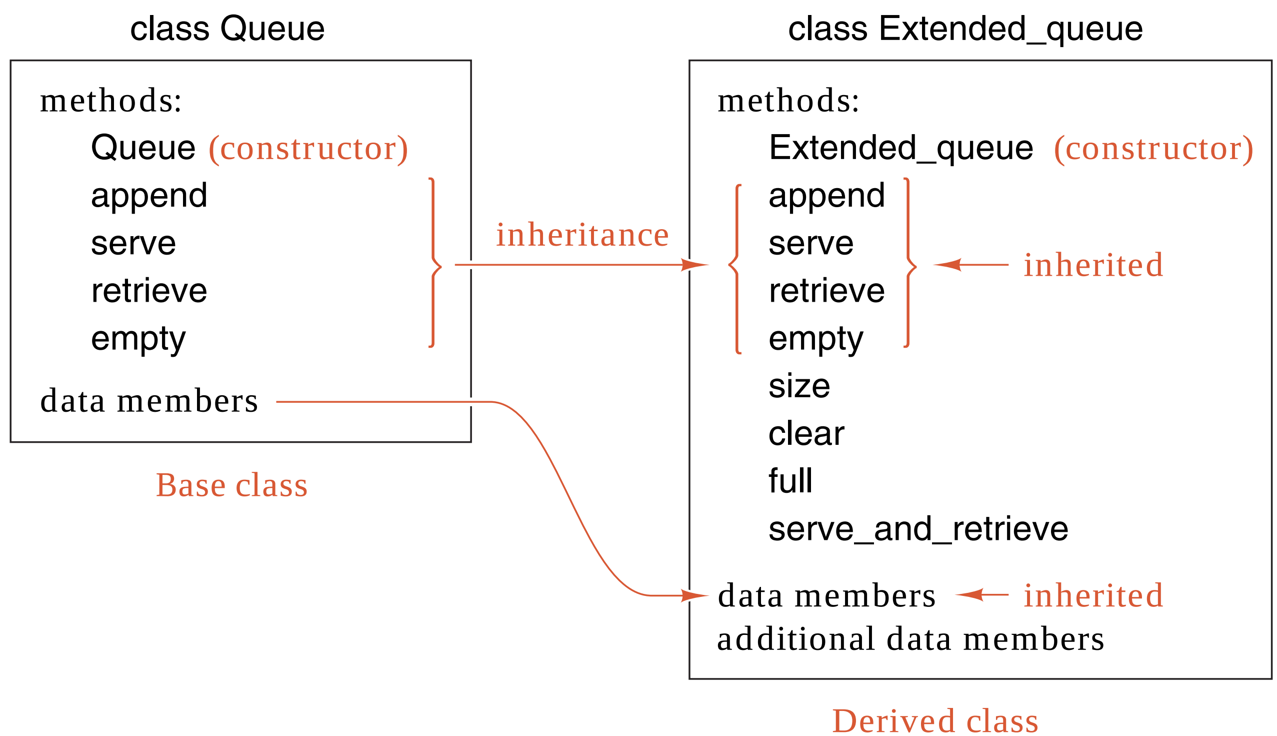
Post: Return the number of entries in the Extended_queue.

Error_code Extended_queue ::
    serve_and_retrieve(Queue_entry &item);

Post: Return underflow if the Extended_queue is empty. Otherwise remove and copy the item at the front of the Extended_queue to item and return success.

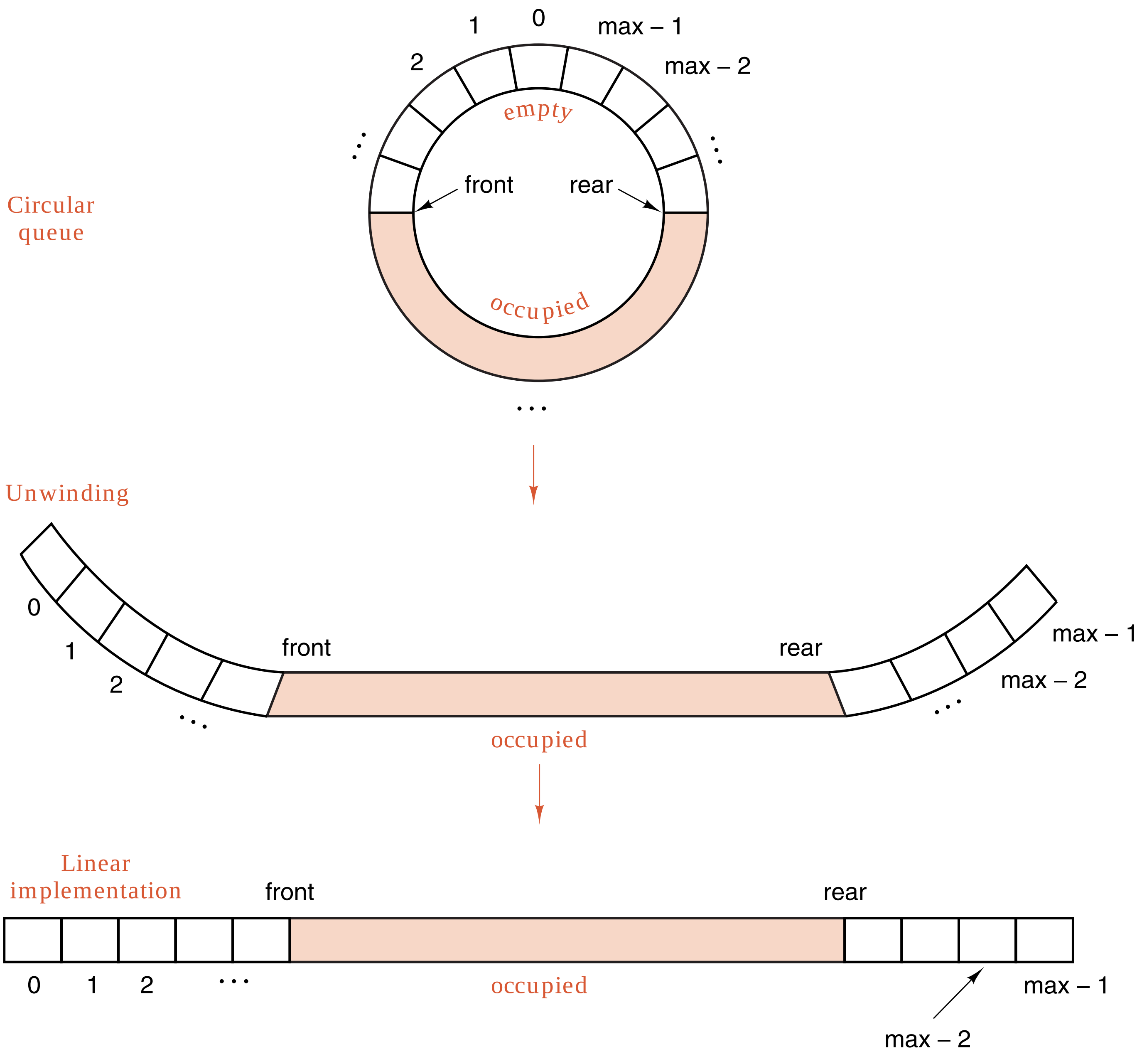


(a) Hierarchy diagram



(b) Derived class Extended_queue from base class Queue

Circular Implementation of Queues

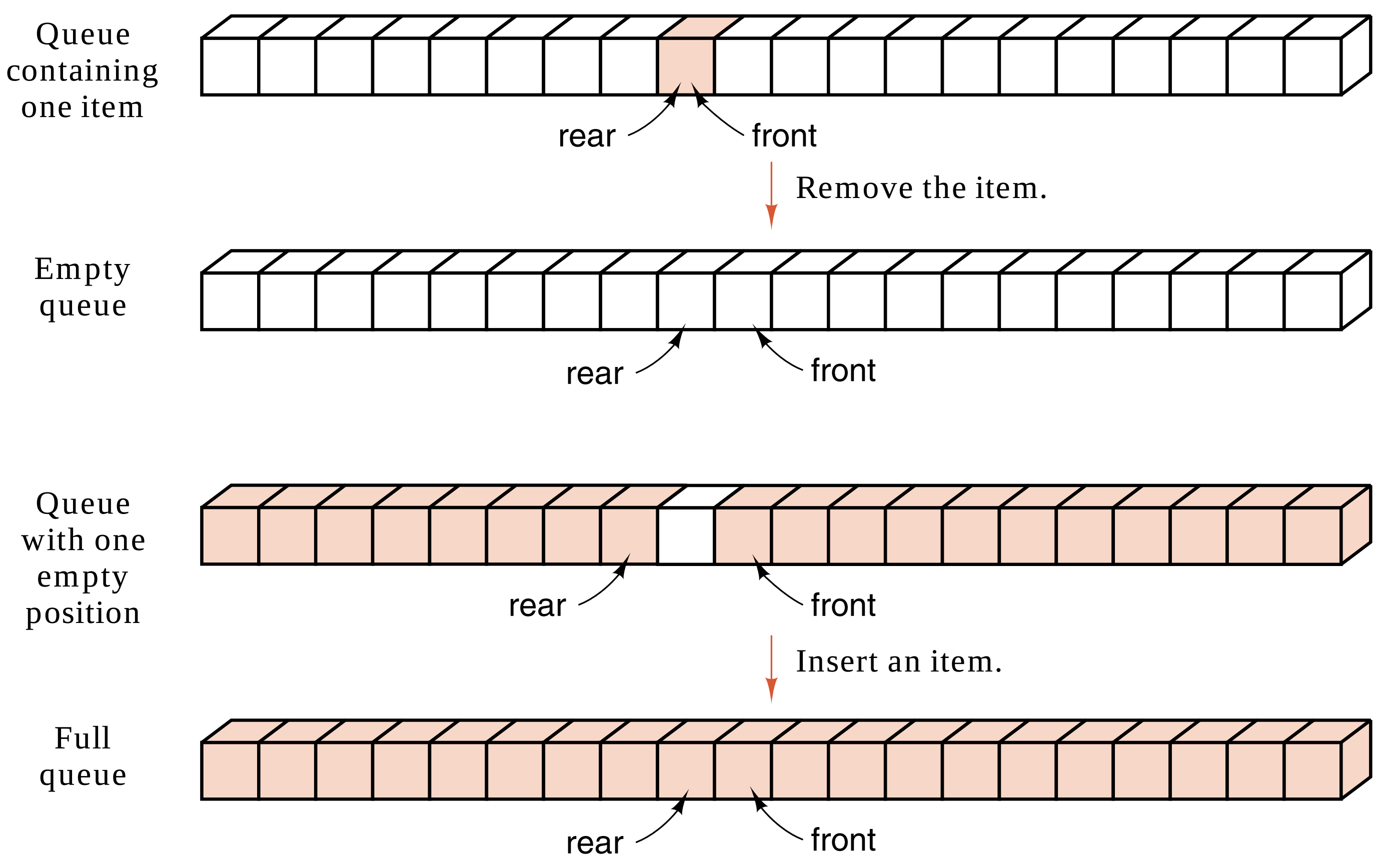


Circular arrays in C++

Equivalent methods to increment an index i in a circular array:

- i = ((i + 1) == max) ? 0 : (i + 1);
- **if** ((i + 1) == max) i = 0; **else** i = i + 1;
- i = (i + 1) % max;

Boundary Conditions



Implementations of Queues

- The *physical model:* a linear array with the front always in the first position and all entries moved up the array whenever the front is deleted.
- A *linear array* with two indices always increasing.
- A *circular array* with front and rear indices and one position left *vacant*.
- A *circular array* with front and rear indices and a Boolean *flag* to indicate fullness (or emptiness).
- A *circular array* with front and rear indices and an *integer counter* of entries.
- A *circular array* with front and rear indices taking *special values* to indicate emptiness.

Programming Precept

Practice information hiding:
Separate the application of data structures from their implementation.

Circular Implementation of Queues in C++

Class definition:

```
const int maxqueue = 10;    //    small value for testing

class Queue {
public:
   Queue( );
   bool empty( ) const;
   Error_code serve( );
   Error_code append(const Queue_entry &item);
   Error_code retrieve(Queue_entry &item) const;
protected:
   int count;
   int front, rear;
   Queue_entry entry[maxqueue];
};
```

Initialization:

```
Queue :: Queue( )
/* Post: The Queue is initialized to be empty. */
{
   count = 0;
   rear = maxqueue − 1;
   front = 0;
}

bool Queue :: empty( ) const
/* Post: Return true if the Queue is empty, otherwise return false. */
{
   return count == 0;
}
```

Basic Queue Methods

```
Error_code Queue :: append(const Queue_entry &item)
/* Post: item is added to the rear of the Queue. If the Queue is full return an
         Error_code of overflow and leave the Queue unchanged. */
{
   if (count >= maxqueue) return overflow;
   count++;
   rear = ((rear + 1) == maxqueue) ? 0 : (rear + 1);
   entry[rear] = item;
   return success;
}

Error_code Queue :: serve( )
/* Post: The front of the Queue is removed. If the Queue is empty return an Er-
         ror_code of underflow. */
{
   if (count <= 0) return underflow;
   count−−;
   front = ((front + 1) == maxqueue) ? 0 : (front + 1);
   return success;
}

Error_code Queue :: retrieve(Queue_entry &item) const
/* Post: The front of the Queue retrieved to the output parameter item. If the Queue
         is empty return an Error_code of underflow. */
{
   if (count <= 0) return underflow;
   item = entry[front];
   return success;
}
```

Demonstration and Testing

```
int main( )
/* Post: Accepts commands from user as a menu-driven demonstration program for the
         class Extended_queue.
   Uses: The class Extended_queue; the functions introduction, get_command,
         and do_command. */

{
   Extended_queue test_queue;
   introduction( );
   while (do_command(get_command( ), test_queue));
}
```

void introduction( )

Post: Writes out an introduction and instructions for the user.

char get_command( )

Post: Gets a valid command from the user and, after converting it to lower case if necessary, returns it.

bool do_command(**char** c, Extended_queue &test_queue)

Pre: c represents a valid command.

Post: Performs the given command c on the Extended_queue test_queue. Returns **false** if c == ′q′, otherwise returns **true**.

Uses: The **class** Extended_queue.

List of Commands

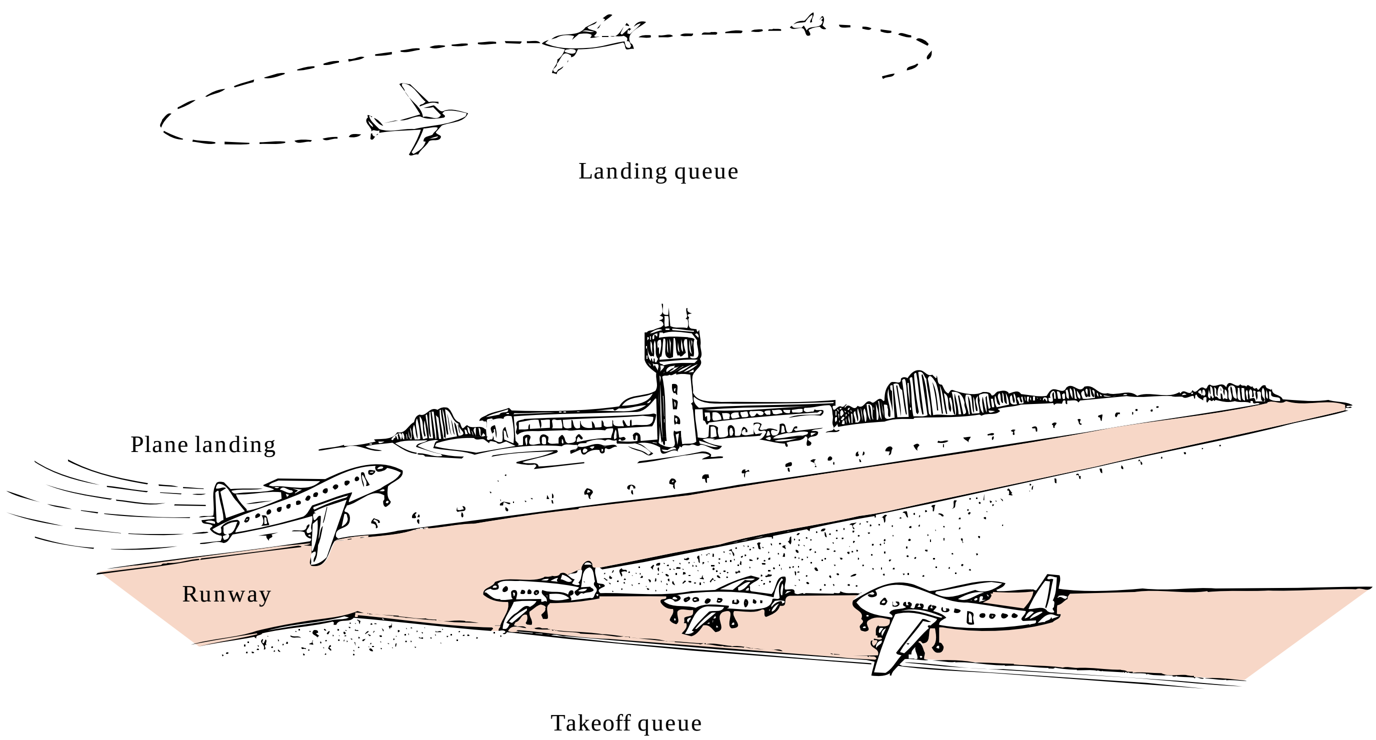
```
void help( )
/* Post:  A help screen for the program is printed, giving the meaning of each command
          that the user may enter.  */

{
  cout << endl
    << "This program allows the user to enter one command" << endl
    << "(but only one) on each input line." << endl
    << "For example, if the command S is entered, then" << endl
    << "the program will serve the front of the queue." << endl
    << endl

    << " The valid commands are:" << endl
    << "A — Append the next input character to the extended queue"
    << endl
    << "S — Serve the front of the extended queue" << endl
    << "R — Retrieve and print the front entry." << endl
    << "# — The current size of the extended queue" << endl
    << "C — Clear the extended queue (same as delete)" << endl
    << "P — Print the extended queue" << endl
    << "H — This help screen" << endl
    << "Q — Quit" << endl

    << "Press < Enter >  to continue." << flush;

  char c;
  do {
    cin.get(c);
  } while (c != '\n');
}
```

Performing a Command

```
bool do_command(char c, Extended_queue &test_queue)
/* Pre:  c represents a valid command.
   Post: Performs the given command c on the Extended_queue test_queue. Re-
         turns false if c == 'q', otherwise returns true.
   Uses: The class Extended_queue. */

{
   bool continue_input = true;
   Queue_entry x;

   switch (c) {
   case 'r':
      if (test_queue.retrieve(x) == underflow)
         cout << "Queue is empty." << endl;
      else
         cout << endl
              << "The first entry is: " << x
              << endl;
      break;

   case 'q':
      cout << "Extended queue demonstration finished." << endl;
      continue_input = false;
      break;

   //  Additional cases will cover other commands.

   }
   return continue_input;
}
```

Application of Queues: Simulation of an Airport

Simulation is the use of one system to imitate the behavior of another system. A ***computer simulation*** is a program to imitate the behavior of the system under study.



1. The same runway is used for both landings and takeoffs.
2. One plane can land or take off in a unit of time, but not both.
3. A random number of planes arrive in each time unit.
4. A plane waiting to land goes before one waiting to take off.
5. The planes that are waiting are kept in queues `landing` and `takeoff`, both of which have a strictly limited size.

```
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++) {
    int number_arrivals = variable.poisson(arrival_rate);
    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);
    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)) {
    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);
}
```

The Runway Class Specification

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

The Plane Class Specification

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

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);
}
```

Runway Methods

```
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;
}

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;
}
```

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;
}
```

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 default
         values. */

{
   flt_num = -1;
   clock_start = -1;
   state = null;
}
```

Plane Methods

```
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;
}
```

```
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;
}
```

```
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;
}
```

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;
}
```

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.