```
            // any unexplored pixels in component?
            if (q.empty()) break;
            here = q.front(); // a component pixel
            q.pop();
         }

      } // end of if, for c, and for r
}
```

Program 9.9 Component labeling (concluded)

9.5.4 Machine Shop Simulation

Problem Description

A machine shop (or factory or plant) comprises m machines or workstations. The machine shop works on jobs, and each job comprises several tasks. Each machine can process one task of one job at any time, and different machines perform different tasks. Once a machine begins to process a task, it continues processing that task until the task completes.

Example 9.2 A sheet metal plant might have one machine (or station) for each of the following tasks: design; cut the sheet metal to size; drill holes; cut holes; trim edges; shape the metal; and seal seams. Each of these machines/stations can work on one task at a time.

Each job includes several tasks. For example, to fabricate the heating and air-conditioning ducts for a new house, we would need to spend some time in the design phase, and then some time cutting the sheet metal stock to the right size pieces. We need to drill or cut the holes (depending on their size), shape the cut pieces into ducts, seal the seams, and trim any rough edges. ■

For each task of a job, there is a task time (i.e., how long does it take) and a machine on which it is to be performed. The tasks of a job are to be performed in a specified order. So a job goes first to the machine for its first task. When this first task is complete, the job goes to the machine for its second task, and so on until its last task completes. When a job arrives at a machine, the job may have to wait because the machine might be busy. In fact, several jobs may already be waiting for that machine.

Each machine in our machine shop can be in one of three states: active, idle, and change over. In the active state the machine is working on a task of some job; in the idle state it is doing nothing; and in the change-over state the machine has completed a task and is preparing for a new task. In the change-over state, the machine operator might, for example, clean the machine, put away tools used for the

last task, and take a break. The time each machine must spend in its change-over state depends on the machine.

When a machine becomes available for a new job, it will need to pick one of the waiting jobs to work on. We assume that each machine serves its waiting jobs in a FIFO manner, and so the waiting jobs at each machine form a (FIFO) queue. Other assumptions for the selection of the next job are possible. For example, the next job may be selected by priority. Each job has a priority, and when a machine becomes free, the waiting job with highest priority is selected.

The time at which a job's last task completes is called its **finish time**. The **length** of a job is the sum of its task times. If a job of length l arrives at the machine shop at time 0 and completes at time f, then it must have spent exactly $f - l$ amount of time waiting in machine queues. To keep customers happy, it is desirable to minimize the time a job spends waiting in machine queues. Machine shop performance can be improved if we know how much time jobs spend waiting and which machines are contributing most to this wait.

How the Simulation Works

When simulating a machine shop, we follow the jobs from machine to machine without physically performing the tasks. We simulate time by using a simulated clock that is advanced each time a task completes or a new job enters the machine shop. As tasks complete, new tasks are scheduled. Each time a task completes or a new job enters the shop, we say that an **event** has occurred. In addition, a **start event** initiates the simulation. When two or more events occur at the same time, we arbitrarily order these events. Figure 9.15 describes how a simulation works.

Example 9.3 Consider a machine shop that has $m = 3$ machines and $n = 4$ jobs. We assume that all four jobs are available at time 0 and that no new jobs become available during the simulation. The simulation will continue until all jobs have completed.

The three machines, $M1$, $M2$, and $M3$, have a change-over time of 2, 3, and 1, respectively. So when a task completes, machine 1 must wait two time units before starting another, machine 2 must wait three time units, and machine 3 must wait one time unit. Figure 9.16(a) gives the characteristics of the four jobs. Job 1, for example, has three tasks. Each task is specified as a pair of the form (machine, time). The first task of job 1 is to be done on $M1$ and takes two time units, the second is to be done on $M2$ and takes four time units, the third is to be done on $M1$ and takes one time unit. The job lengths (i.e., the sum of their task times) are 7, 6, 8, and 4, respectively.

Figure 9.16(b) shows the machine shop simulation. Initially, the four jobs are placed into queues corresponding to their first tasks. The first task for jobs 1 and 3 are to be done on $M1$, so these jobs are placed on the queue for $M1$. The first tasks for jobs 2 and 4 are to be done on $M3$. Consequently, these jobs begin on the queue for $M3$. The queue for $M2$ is empty. At the start all three machines are idle.

```
// initialize
input the data;
create the job queues at each machine;
schedule first job in each machine queue;

// do the simulation
while (an unfinished job remains)
{
    determine the next event;
    if (the next event is the completion of a machine change over)
       schedule the next job (if any) from this machine's queue;
    else
      {// a job task has completed
       put the machine that finished the job task into its change-over state;
       move the job whose task has finished to the machine for its next task
       (if any);
      }
}
```

Figure 9.15 The mechanics of simulation

We use the symbol I to indicate that the machines have no active job at this time. Since no machine is active, the time at which they will finish their current active task is undefined and denoted by the symbol L (large time).

The simulation begins at time 0. That is, the first event, the start event, occurs at time 0. At this time the first job in each machine queue is scheduled on the corresponding machine. Job 1's first task is scheduled on $M1$, and job 2's first task on $M3$. The queue for $M1$ now contains job 3 only, while that for $M3$ contains job 4 only. The queue for $M2$ remains empty. Job 1 becomes the active job on $M1$, and job 2 the active job on $M3$. $M2$ remains idle. The finish time for $M1$ becomes 2 (current time of 0 plus task time of 2), and the finish time for $M3$ becomes 4.

The next event occurs at time 2. This time is determined by finding the minimum of the machine finish times. At time 2 machine $M1$ completes its active task. This task is a job 1 task. Job 1 is moved to machine $M2$ for the next task. Since $M2$ is idle, the processing of job 1's second task begins immediately. This task will complete at time 6 (current time of 2 plus task time of 4). $M1$ goes into its change-over state and will remain in this state for two time units. Its active job is set to C (change over), and its finish time is set to 4.

At time 4 both $M1$ and $M3$ complete their active tasks. As machine $M1$ completes a change-over task, that machine begins a new job; selecting the first job, job 3, from its queue. Since the task length for job 3's next task is 4, the task will complete at time 8 and the finish time for $M1$ becomes 8. The next task for job 2,

Job#	#Tasks	Tasks	Length
1	3	(1,2) (2,4) (1,1)	7
2	2	(3,4) (1,2)	6
3	2	(1,4) (2,4)	8
4	2	(3,1) (2,3)	4

(a) Job characteristics

Time	Machine Queues			Active Jobs			Finish Times		
	$M1$	$M2$	$M3$	$M1$	$M2$	$M3$	$M1$	$M2$	$M3$
Init	1,3	–	2,4	I	I	I	L	L	L
0	3	–	4	1	I	2	2	L	4
2	3	–	4	C	1	2	4	6	4
4	2	–	4	3	1	C	8	6	5
5	2	–	–	3	1	4	8	6	6
6	2,1	4	–	3	C	C	8	9	7
7	2,1	4	–	3	C	I	8	9	L
8	2,1	4,3	–	C	C	I	10	9	L
9	2,1	3	–	C	4	I	10	12	L
10	1	3	–	2	4	I	12	12	L
12	1	3	–	C	C	I	14	15	L
14	–	3	–	1	C	I	15	15	L
15	–	–	–	C	3	I	17	19	L
16	–	–	–	C	3	I	17	19	L
17	–	–	–	I	3	I	L	19	L

(b) Simulation

Job#	Finish Time	Wait Time
1	15	8
2	12	6
3	19	11
4	12	8
Total	58	33

(c) Finish and wait times

Figure 9.16 Machine shop simulation example

which just completed its first task on machine $M3$, needs to be done on $M1$. Since $M1$ is busy, job 2 is added to $M1$'s job queue. $M3$ moves into its change-over state and completes this change-over task at time 5. You should now be able to follow

the remaining sequence of events.

Figure 9.16(c) gives the finish and wait times. Since the length of job 2 is 6 and its finish time 12, job 2 must have spent a total of $12 - 6 = 6$ time units waiting in machine queues. Similarly, job 4 must have spent $12 - 4 = 8$ time units waiting in queues.

We may determine the distribution of the 33 units of total wait time across the three machines. For example, job 4 joined the queue for $M3$ at time 0 and did not become active until time 5. So this job waited at $M3$ for five time units. No other job experienced a wait at $M3$. The total wait time at $M3$ was, therefore, five time units. Going through Figure 9.16(b), we can compute the wait times for $M1$ and $M2$. The numbers are 18 and 10, respectively. As expected the sum of the job wait times (33) equals the sum of the machine wait times. ■

Benefits of Simulating a Machine Shop

Why do we want to simulate a machine shop? Here are some reasons:

- By simulating the shop, we can identify bottleneck machines/stations. If we determine that the paint station is going to be a bottleneck for the current mix of jobs, we can increase the number of paint stations in operation for the next few shifts. Similarly, if our simulation determines that the wait time at the drill station will be excessive in the next shift, we can schedule more drill station operators and put more drilling machines to work. Therefore, the simulator can be used for short-term operator-scheduling decisions.

- Using a machine shop simulator, we can answer questions such as, How is average wait time affected if we replace a certain machine with a more expensive but more effective machine? So the simulator can be used to help make expansion/modernization decisions at the factory.

- When customers arrive at the plant, they would like a fairly accurate estimate of when their jobs will complete. Such an estimate may be obtained by using a machine shop simulator.

High-Level Simulator Design

In designing our simulator, we will assume that all jobs are available initially (i.e., no jobs enter the shop during the simulation). Further, we assume that the simulation is to be run until all jobs complete.

The simulator is implemented as the class `machineShopSimulator`. Since the simulator is a fairly complex program, we break it into modules. The tasks to be performed by the simulator are input the data and put the jobs into the queues for their first tasks; perform the start event (i.e., do the initial loading of jobs onto the machines); run through all the events (i.e., perform the actual simulation); and

output the machine wait times. We will have one C++ function for each task. Program 9.10 gives the main function. The variable `largeTime` is a global variable that denotes a time that is larger than any permissible simulated time; that is all tasks of all jobs must complete before the time `largeTime`.

```
void main()
{
   inputData();          // get machine and job data
   startShop();          // initial machine loading
   simulate();           // run all jobs through shop
   outputStatistics();   // output machine wait times
}
```

Program 9.10 Main function for machine shop simulation

The Struct Task

Before we can develop the code for the four functions invoked by Program 9.10, we must develop representations for the data objects that are needed. These objects include tasks, jobs, machines, and an event list. We define a struct for the first three of these data object and a class for the third.

Each task has two components: `machine` (the machine on which it is to be performed) and `time` (the time needed to complete the task). Program 9.11 gives the struct `task`. Since machines are assumed to be have integer labels, `machine` is of type `int`. We will assume that all times are integral.

```
struct task
{
   int machine;
   int time;

   task(int theMachine = 0, int theTime = 0)
   {
      machine = theMachine;
      time = theTime;
   }
};
```

Program 9.11 The struct `task`

The Struct job

Each job has a list of associated tasks that are performed in list order. Consequently, the task list may be represented as a queue **taskQ**. To determine the total wait time experienced by a job, we need to know its length and finish time. The finish time is determined by the event clock, while the job length is the sum of task times. To determine a job's length, we associate a data member **length** with it. Program 9.12 gives the struct **job**.

```
struct job
{
   arrayQueue<task> taskQ;    // this job's tasks
   int length;                // sum of scheduled task times
   int arrivalTime;           // arrival time at current queue
   int id;                    // job identifier

   job(int theId = 0)
   {
      id = theId;
      length = 0;
      arrivalTime = 0;
   }

   void addTask(int theMachine, int theTime)
   {
      task theTask(theMachine, theTime);
      taskQ.push(theTask);
   }

   int removeNextTask()
   {// Remove next task of job and return its time.
    // Also update length.

      int theTime =  taskQ.front().time;
      taskQ.pop();
      length += theTime;
      return theTime;
   }
};
```

Program 9.12 The struct **job**

The data member **arrivalTime** records the time at which a job enters its current

machine queue and determines the time the job waits in this queue. The job identifier is stored in `id` and is used only when outputting the total wait time encountered by this job.

The method `addTask` adds a task to the job's task queue. The task is to be performed on machine `theMachine` and takes `theTime` time. This method is used only during data input. The method `removeNextTask` is used when a job is moved from a machine queue to active status. At this time the job's first task is removed from the task queue (the task queue maintains a list of tasks yet to be scheduled on machines), the job length is incremented by the task time, and the task time is returned. The data member `length` becomes equal to the job length when we schedule the last task for the job.

The Struct `machine`

Each machine has a change-over time, an active job, and a queue of waiting jobs. Since each job can be in at most one machine queue at any time, the total space needed for all queues is bounded by the number of jobs. However, the distribution of jobs over the machine queues changes as the simulation proceeds. It is possible to have a few very long queues at one time. These queues might become very short later, and some other queues become long. By using linked queues, we limit the space required for the machine queues to that required for n nodes where n is the number of jobs.

Program 9.13 gives the struct `machine`. The data members `jobQ`, `changeTime`, `totalWait`, `numTasks`, and `activeJob`, respectively, denote the queue of waiting jobs, the change-over time for the machine, the total time jobs have spent waiting at this machine, the number of tasks processed by the machine, and the currently active job. The currently active job is `NULL` whenever the machine is idle or in its change-over state.

The Class `eventList`

We store the finish times of all machines in an event list. To go from one event to the next, we need to determine the minimum of these finish times. Our simulator also needs an operation that sets the finish time of a particular machine. This operation has to be done each time a new job is scheduled on a machine. When a machine becomes idle, its finish time is set to the large number `largeTime`. Program 9.14 gives the class `eventList` that implements the event list as a one-dimensional array `finishTime`, with `finishTime[p]` being the finish time of machine `p`.

The method `nextEventMachine` returns the machine that completes its active task first. The time at which machine `p` finishes its active task can be determined by invoking the method `nextEventTime(p)`. For a machine shop with m machines, it takes $\Theta(m)$ time to find the minimum of the finish times, so the complexity of `nextEventMachine` is $\Theta(m)$. The method to set the finish time of a machine, `setFinishTime`, runs in $\Theta(1)$ time. In Chapter 13 we will see two data

```
struct machine
{
   arrayQueue<job*> jobQ;
                    // queue of waiting jobs for this machine
   int changeTime;  // machine change-over time
   int totalWait;   // total delay at this machine
   int numTasks;    // number of tasks processed on this machine
   job* activeJob;  // job currently active on this machine

   machine()
   {
      totalWait = 0;
      numTasks = 0;
      activeJob = NULL;
   }
};
```

Program 9.13 The struct `machine`

structures—heaps and leftist trees—that may also represent an event list. When we use either of these data structures, the complexity of both `nextEventMachine` and `setFinishTime` becomes $O(\log m)$. If the total number of tasks across all jobs is $numTasks$, then, in a successful simulation run, our simulator will invoke `nextEventMachine` and `setFinishTime` $\Theta(numTasks)$ times each. Using the event list implementation of Program 9.14, these invocations take a total of $\Theta(numTasks * m)$ time; using one of the data structures of Chapter 13, the invocations take $O(numTasks * \log m)$ time. Even though the data structures of Chapter 13 are more complex, they result in a faster simulation when the number of machines m is suitably large.

Global Variables

Program 9.15 gives the global variables used by our code. The significance of most of these variables is self-evident. `timeNow` is the simulated clock and records the current time. Each time an event occurs, it is updated to the event time. `largeTime` is a time that exceeds the finish time of the last job and denotes the finish time of an idle machine.

The Function `inputData`

The code for the function `inputData` (Program 9.16) begins by inputting the number of machines and jobs in the shop. Next we create the initial event list `eList`,

```
class eventList
{
   public:
      eventList(int theNumMachines, int theLargeTime)
      {// Initialize finish times for m machines.
         if (theNumMachines < 1)
            throw illegalParameterValue
                  ("number of machines must be >= 1");
         numMachines = theNumMachines;
         finishTime = new int [numMachines + 1];

         // all machines are idle, initialize with
         // large finish time
         for (int i = 1; i <= numMachines; i++)
            finishTime[i] = theLargeTime;
      }

      int nextEventMachine()
      {// Return machine for next event.

         // find first machine to finish, this is the
         // machine with smallest finish time
         int p = 1;
         int t = finishTime[1];
         for (int i = 2; i <= numMachines; i++)
            if (finishTime[i] < t)
            {// i finishes earlier
               p = i;
               t = finishTime[i];
            }
         return p;
      }

      int nextEventTime(int theMachine)
      {return finishTime[theMachine];}

      void setFinishTime(int theMachine, int theTime)
      {finishTime[theMachine] = theTime;}
   private:
      int* finishTime;    // finish time array
      int numMachines;    // number of machines
};
```

Program 9.14 The class eventList

```
// global variables
int timeNow;              // current time
int numMachines;          // number of machines
int numJobs;              // number of jobs
eventList* eList;         // pointer to event list
machine* mArray;          // array of machines
int largeTime = 10000;    // all machines finish before this
```

Program 9.15 Global variables for machine shop simulation

```
void inputData()
{// Input machine shop data.

   cout << "Enter number of machines and jobs" << endl;
   cin >> numMachines >> numJobs;
   if (numMachines < 1 || numJobs < 1)
         throw illegalInputData
               ("number of machines and jobs must be >= 1");

   // create event and machine queues
   eList = new eventList(numMachines, largeTime);
   mArray = new machine [numMachines + 1];

   // input the change-over times
   cout << "Enter change-over times for machines" << endl;
   int ct;
   for (int j = 1; j <= numMachines; j++)
   {
      cin >> ct;
      if (ct < 0)
         throw illegalInputData("change-over time must be >= 0");
      mArray[j].changeTime = ct;
   }
```

Program 9.16 Code to input machine shop data (continues)

with finish times equal to **largeTime** for each machine, and the array **mArray** of machines. Then we input the change-over times for the machines. Next we input the jobs one by one. For each job we first input the number of tasks it has, and then we input the tasks as pairs of the form (machine, time). The machine for the

```
   // input the jobs
   job* theJob;
   int numTasks, firstMachine, theMachine, theTaskTime;
   for (int i = 1; i <= numJobs; i++)
   {
      cout << "Enter number of tasks for job " << i << endl;
      cin >> numTasks;
      firstMachine = 0;   // machine for first task
      if (numTasks < 1)
         throw illegalInputData("each job must have > 1 task");

      // create the job
      theJob = new job(i);
      cout << "Enter the tasks (machine, time)"
           << " in process order" << endl;
      for (int j = 1; j <= numTasks; j++)
      {// get tasks for job i
         cin >> theMachine >> theTaskTime;
         if (theMachine < 1 || theMachine > numMachines
             || theTaskTime < 1)
         throw illegalInputData("bad machine number or task time");
         if (j == 1)
            firstMachine = theMachine;    // job's first machine
         theJob->addTask(theMachine, theTaskTime);  // add to
      }                                             // task queue
      mArray[firstMachine].jobQ.push(theJob);
   }
}
```

Program 9.16 Code to input machine shop data (concluded)

first task of the job is recorded in the variable `firstMachine`. When all tasks of a job have been input, the job is added to the queue for the first task's machine.

The Functions startShop and changeState

To start the simulation, we need to move the first job from each machine's job queue to the machine and commence processing. Since each machine is initialized in its idle state, we perform the initial loading in the same way as we change a machine from its idle state, which may happen during simulation, to an active state. The method `changeState(i)` performs this change over for machine `i`. The method to start the shop, Program 9.17, needs merely invoke `changeState` for each machine.

```
void startShop()
{// Load first jobs onto each machine.
   for (int p = 1; p <= numMachines; p++)
      changeState(p);
}
```

Program 9.17 Initial loading of machines

Program 9.18 gives the code for `changeState`. If machine `theMachine` is idle or in its change-over state, `changeState` returns `NULL`. Otherwise, it returns the job that `theMachine` has been working on. Additionally, `changeState(theMachine)` changes the state of machine `theMachine`. If machine `theMachine` was previously idle or in its change-over state, then it begins to process the next job on its queue. If that queue is empty, the machine's new state is "idle." If machine `theMachine` was previously processing a job, machine `theMachine` moves into its change-over state.

If `mArray[theMachine].activeJob` is `NULL`, then machine `theMachine` is either in its idle or change-over state; the job, `lastJob`, to return is `NULL`. If the job queue is empty, the machine moves into its idle state and its finish time is set to `largeTime`. If its job queue is not empty, the first job is removed from the queue and becomes machine `theMachine`'s active job. The time this job has spent waiting in machine `theMachine`'s queue is added to the total wait time for this machine, and the number of tasks processed by the machine incremented by 1. Next the task that this machine is going to work on is deleted from the job's task list, and the finish time of the machine is set to the time at which the new task will complete.

If `mArray[theMachine].activeJob` is not `NULL`, the machine has been working on a job whose task has just completed. Since this job is to be returned, we save it in `lastJob`. The machine should now move into its change-over state and remain in that state for `changeTime` time units.

The Functions `simulate` and `moveToNextMachine`

The function `simulate`, Program 9.19, cycles through all shop events until the last job completes. `numJobs` is the number of incomplete jobs, so the `while` loop of Program 9.19 terminates when no incomplete jobs remain. In each iteration of the `while` loop, the time for the next event is determined, and the clock time `timeNow` updated to this event time. A change-job operation is performed on the machine `nextToFinish` on which the event occurred. If this machine has just finished a task of a job (`theJob` is not `NULL`), job `theJob` moves to the machine on which its next task is to be performed. The function `moveToNextMachine` performs this move. If there is no next task for job `theJob`, the job has completed, function `moveToNextMachine` returns `false`, and `numJobs` is decremented by 1.

```
job* changeState(int theMachine)
{// Task on theMachine has finished, schedule next one.
 // Return last job run on this machine.
   job* lastJob;
   if (mArray[theMachine].activeJob == NULL)
   {// in idle or change-over state
      lastJob = NULL;
      // wait over, ready for new job
      if (mArray[theMachine].jobQ.empty()) // no waiting job
            eList->setFinishTime(theMachine, largeTime);
      else
      {// take job off the queue and work on it
         mArray[theMachine].activeJob =
               mArray[theMachine].jobQ.front();
         mArray[theMachine].jobQ.pop();
         mArray[theMachine].totalWait +=
              timeNow - mArray[theMachine].activeJob->arrivalTime;
         mArray[theMachine].numTasks++;
         int t = mArray[theMachine].activeJob->removeNextTask();
         eList->setFinishTime(theMachine, timeNow + t);
      }
   }
   else
   {// task has just finished on theMachine
    // schedule change-over time
      lastJob = mArray[theMachine].activeJob;
      mArray[theMachine].activeJob = NULL;
      eList->setFinishTime(theMachine, timeNow +
                           mArray[theMachine].changeTime);
   }

   return lastJob;
}
```

Program 9.18 Code to change the active job at a machine

The function `moveToNextMachine` (Program 9.20) first checks to see whether any unprocessed tasks remain for the job `theJob`. If not, the job has completed and its finish time and wait time are output. The method returns `false` to indicate there was no next machine for this job.

When the job `theJob` to be moved has a next task, the machine `p` for this task

```
void simulate()
{// Process all jobs to completion.
   while (numJobs > 0)
   {// at least one job left
      int nextToFinish = eList->nextEventMachine();
      timeNow = eList->nextEventTime(nextToFinish);
      // change job on machine nextToFinish
      job* theJob = changeState(nextToFinish);
      // move theJob to its next machine
      // decrement numJobs if theJob has finished
      if (theJob != NULL && !moveToNextMachine(theJob))
         numJobs--;
   }
}
```

Program 9.19 Run all jobs through their machines

is determined and the job is added to this machine's queue of waiting jobs. In case machine `p` is idle, `changeState` is invoked to change the state of machine `p` so that machine `p` begins immediately to process the next task of `theJob`.

The Function outputStatistics

Since both the time at which a job finishes and the time a job spends waiting in machine queues are output by `moveToNextMachine`, `outputStatistics` needs to output only the time at which the machine shop completes all jobs (this time is also the time at which the last job completed and has been output by `moveToNextMachine`) and the statistics (total wait time and number of tasks processed) for each machine. Program 9.21 gives the code.

EXERCISES

19. Which applications from this section can use a stack instead of a queue without affecting the correctness of the program?

20. (a) You have a railroad shunting yard with three shunting tracks that operate as queues. The initial ordering of cars is 3, 1, 7, 6, 2, 8, 5, 4. Draw figures similar to Figures 9.11 to show the configuration of the shunting tracks, the input track, and the output track following each car move made by the solution of Section 9.5.1.

    (b) Do part (a) for the case when the number of shunting tracks is 2.

```
bool moveToNextMachine(job* theJob)
{// Move theJob to machine for its next task.
 // Return false iff no next task.

   if (theJob->taskQ.empty())
   {// no next task
      cout << "Job " << theJob->id << " has completed at "
           << timeNow << " Total wait was "
           << (timeNow - theJob->length) << endl;
      return false;
   }
   else
   {// theJob has a next task
    // get machine for next task
      int p = theJob->taskQ.front().machine;
      // put on machine p's wait queue
      mArray[p].jobQ.push(theJob);
      theJob->arrivalTime = timeNow;
      // if p idle, schedule immediately
      if (eList->nextEventTime(p) == largeTime)
         // machine is idle
         changeState(p);

      return true;
   }
}
```

Program 9.20 Move a job to the machine for the next task

21. Does Program 9.6 successfully rearrange all input car permutations that can be rearranged using k holding tracks that operate as queues? Prove your answer.

22. Rewrite Program 9.6 under the assumption that at most s_i cars can be in holding track i at any time. Reserve the track with smallest s_i for direct input to output moves.

23. Can you eliminate the use of queues and, instead, use the strategy of the second implementation of the railroad car problem when you have to display the state of the holding tracks following each move of a railroad car? Justify your answer.

24. Is it possible to solve the problem of Section 8.5.3 without the use of a stack

```
void outputStatistics()
{// Output wait times at machines.
   cout << "Finish time = " << timeNow << endl;
   for (int p = 1; p <= numMachines; p++)
   {
      cout << "Machine " << p << " completed "
           << mArray[p].numTasks << " tasks" << endl;
      cout << "The total wait time was "
           << mArray[p].totalWait << endl;
      cout <<  endl;
   }
}
```

Program 9.21 Output the wait times at each machine

(see second implementation of Section 9.5.1)? If so, develop and test such a program.

25. Consider the wire-routing grid of Figure 9.13(a). You are to route a wire between (1, 4) and (2, 2). Label all grid positions that are reached in the distance-labeling pass by their distance value. Now use the methodology of the path-identification pass to mark the shortest wire path.

26. Develop a complete C++ program for wire routing. Your program should include a `welcome` method that displays the program name and functionality; a method to input the wire grid size, blocked and unblocked grid positions, and wire endpoints; the method `findPath` (Program 9.8); and a method to output the input grid with the wire path shown. Test your code.

27. In a typical wire-routing application, several wires are routed in sequence. After a path has been found for one wire, the grid positions used by this path are blocked and we proceed to find a path for the next wire. When the array `grid` is overloaded to designate both blocked and unblocked positions as well as distances, we must clean the grid (i.e., set all grid positions that are on the wire path to 1 and all remaining positions with a label > 1 to 0) before we can begin on the next wire. Write a method to clean the grid. Do so by first restoring the grid to its initial state, using a process similar to that used in the distance-labeling pass. Then write code to block the positions on the wire path just found. This way the complexity of the cleanup pass is the same as that of the distance-labeling pass.

28. Develop a complete C++ program for image-component labeling. Your program should include a `welcome` function that displays the program name and