COURSE TECHNOLOGY
CENGAGE Learning™

# Data Structures Using C++ 2E

*Chapter 8*
*Queues*

# Objectives

- Learn about queues
- Examine various queue operations
- Learn how to implement a queue as an array
- Learn how to implement a queue as a linked list
- Discover queue applications
- Become aware of the STL `class queue`
- Queue data structure
  - Elements added at one end (rear), deleted from other end (front)
  - First In First Out (FIFO)
  - Middle elements inaccessible

# Queue Operations

- Two key operations
  - `addQueue`: add to the back of the queue
  - `deleteQueue`: remove from the front of the queue
- Additional operations
  - `initializeQueue`, `isEmptyQueue`, `isFullQueue`, `front`, `back`
- `queueFront`, `queueRear` pointers
  - Keep track of front and rear
- See code on pages 453-454

```
template <class Type>
class queueADT
{
public:
    virtual bool isEmptyQueue() const = 0;
      //Function to determine whether the queue is empty.

    virtual bool isFullQueue() const = 0;
      //Function to determine whether the queue is full.

    virtual void initializeQueue() = 0;
      //Function to initialize the queue to an empty state.

    virtual Type front() const = 0;
      //Function to return the first element of the queue.

    virtual Type back() const = 0;
      //Function to return the last element of the queue.

    virtual void addQueue(const Type& queueElement) = 0;
      //Function to add queueElement to the queue.

    virtual void deleteQueue() = 0;
      //Function to remove the first element of the queue.
};
```

# Implementation of Queues as Arrays

- Four member variables
  - Array to store queue elements
  - Variables `queueFront`, `queueRear`
  - Variable `maxQueueSize`
- Using `queueFront`, `queueRear` to access queue elements
  - `queueFront`: first queue element index
  - `queueRear`: last queue element index
    - `queueFront` changes after each `deleteQueue` operation
    - `queueRear` changes after each `addQueue` operation

```
template <class Type>
class queueType: public queueADT<Type>
{
public:
    const queueType<Type>& operator=(const queueType<Type>&);
      //Overload the assignment operator.

    bool isEmptyQueue() const;
    bool isFullQueue() const;
    void initializeQueue();
    Type front() const;
    Type back() const;

    void addQueue(const Type& queueElement);
      //Function to add queueElement to the queue.

    void deleteQueue();
      //Function to remove the first element of the queue.

    queueType(int queueSize = 100);
    queueType(const queueType<Type>& otherQueue);

    ~queueType();

private:
    int maxQueueSize; //variable to store the maximum queue size
    int count;        //variable to store the number of
                      //elements in the queue
    int queueFront;   //variable to point to the first
                      //element of the queue
    int queueRear;    //variable to point to the last
                      //element of the queue
    Type *list;       //pointer to the array that holds
                      //the queue elements
};
```

# Implementation of Queues as Arrays (cont'd.)

- Execute operation
  - `Queue.addQueue('A');`
- Execute
  - `Queue.addQueue('B');`
  - `Queue.addQueue('C');`
- Execute
  - `Queue.deleteQueue();`

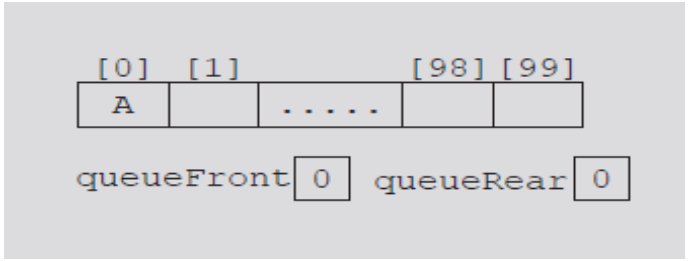



**FIGURE 8-1** Queue after the first addQueue operation

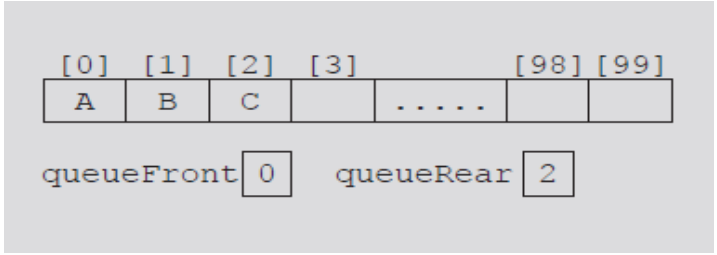



**FIGURE 8-2** Queue after two more `addQueue` operations

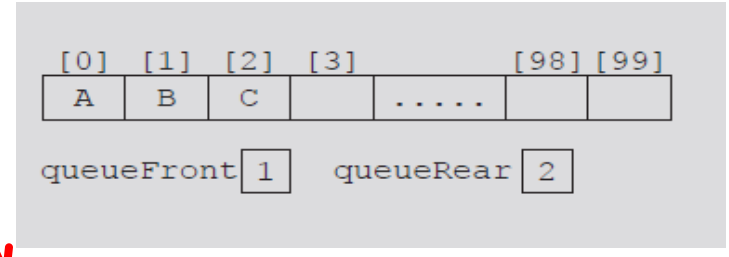



**FIGURE 8-3** Queue after the `deleteQueue` operation

# Implementation of Queues as Arrays (cont'd.)

- Consider the sequence of operations: AAADADADADADADADA...
  - Eventually index `queueRear` points to last array position
    - Looks like a full queue
    - Reality: queue has two or three elements, array empty in the front

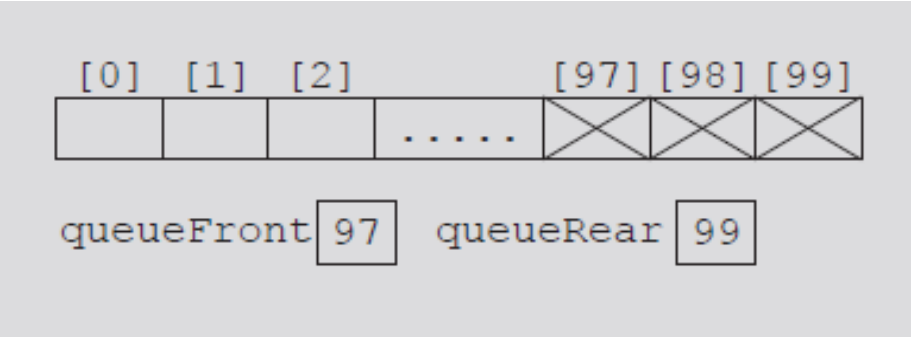



**FIGURE 8-4** Queue after the sequence of operations AAADADADADADA...

# Implementation of Queues as Arrays (cont'd.)

- First solution
  - Upon queue overflow to the rear
    - Check value of `queueFront`
    - If room in front: slide all queue elements toward first array position
  - Works if queue size very small
- Second solution: assume circular array

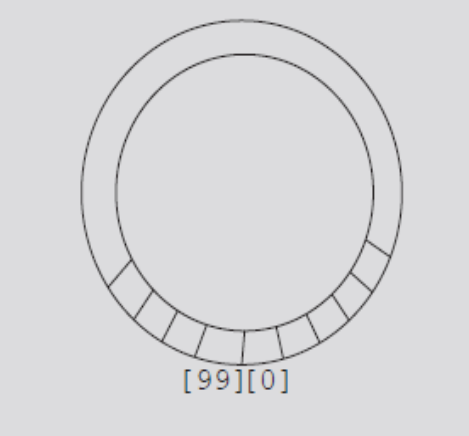



**FIGURE 8-5** Circular queue

# Queue as Circular Array

- addQueue
  - Advances `queueRear` (`queueFront`) to next array position

```
queueRear = (queueRear + 1) % maxQueueSize;
```

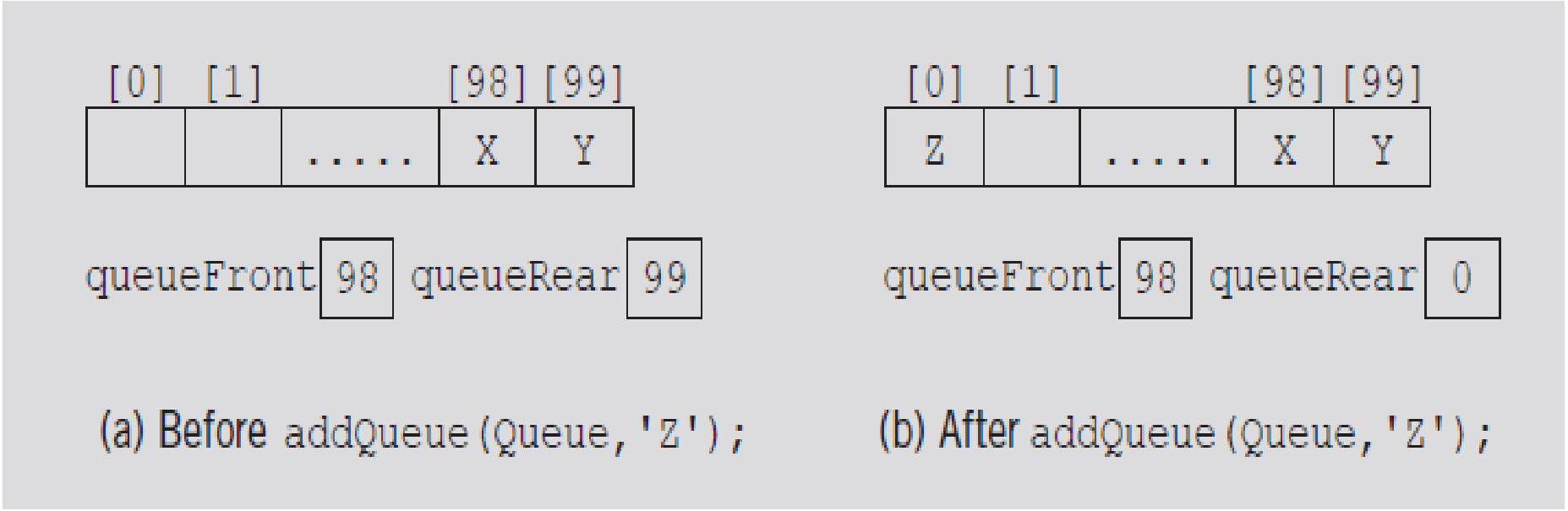



**FIGURE 8-6** Queue before and after the `add` operation

# Queue as Circular Array (cont'd.)

- If `queueRear` < `maxQueueSize` – 1
  - `queueRear` + 1 <= `maxQueueSize` – 1
  - (`queueRear` + 1) % `maxQueueSize` ➔ `queueRear` + 1
- If `queueRear` == `maxQueueSize` – 1
  - `queueRear` + 1 == `maxQueueSize`
  - (`queueRear` + 1) % `maxQueueSize` ➔ 0
- `queueRear` set to zero
  - First array position

# Queue Empty or Full?

- Problematic cases with identical `queueFront`, `queueRear` values
  - Figure 8-7(b) represents an empty queue
  - Figure 8-8(b) represents a full queue

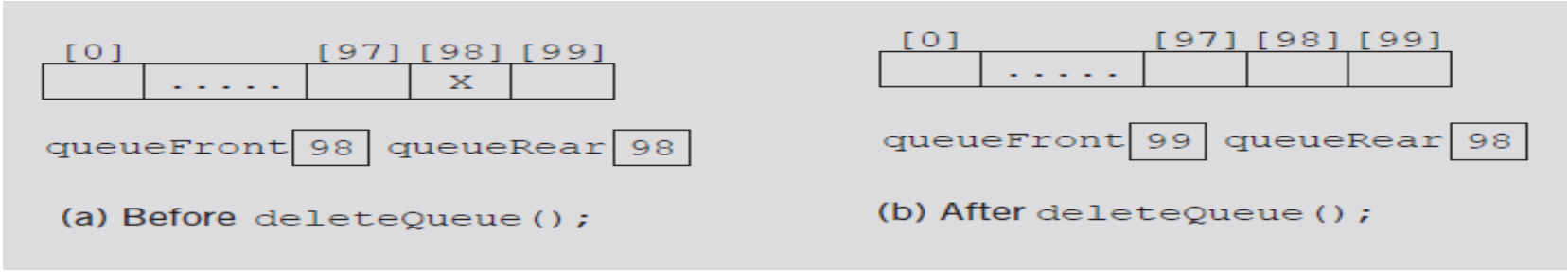



**FIGURE 8-7** Queue before and after the `delete` operation

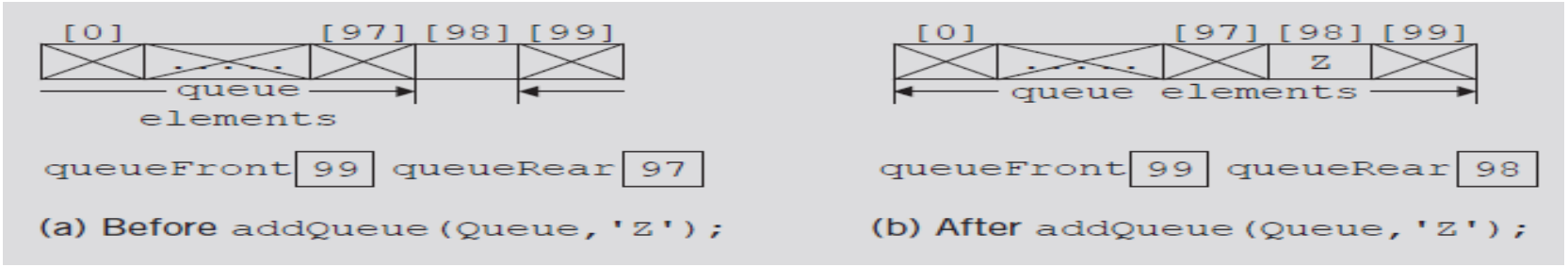



**FIGURE 8-8** Queue before and after the `add` operation

# Queue Empty or Full? (cont'd.)

- First solution: use variable `count`
  - Incremented when new element added
  - Decremented when element removed
  - Functions `initializeQueue`, `destroyQueue` initialize count to zero

- See code on pages 459-460
  - Uses first solution

# Queue Empty or Full? (cont'd.)

- Second solution
  - `queueFront` indicates index of array position *preceding* first element of the queue
  - Assume `queueRear` indicates index of last element
  - Slot indicated by index `queueFront` is reserved
    - If array size is N, max N-1 elements can be stored in queue
  - Queue is empty if `queueFront == queueRear`
  - Queue is full if next available space represents special reserved slot

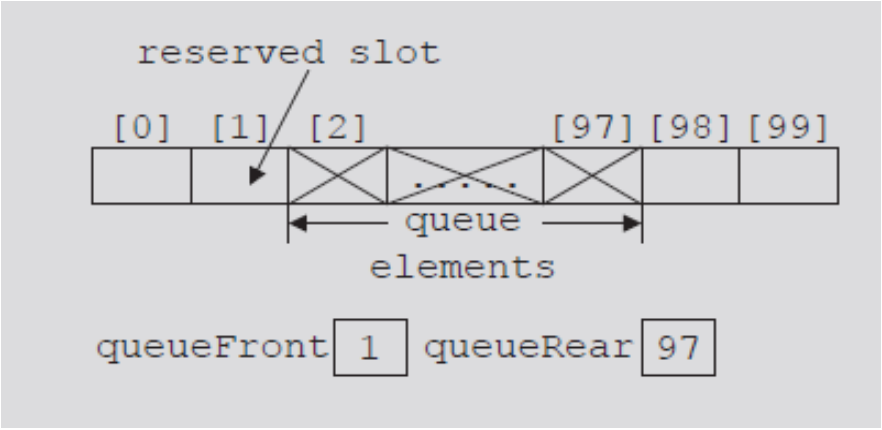



**FIGURE 8-9** Array to store the queue elements with a reserved slot

# Circular Queue w/ count: Empty Queue and Full Queue

- Empty queue
  - If `count == 0`
- Full queue
  - If `count == maxQueueSize`

```
template <class Type>
bool queueType<Type>::isEmptyQueue() const
{
    return (count == 0);
} //end isEmptyQueue

template <class Type>
bool queueType<Type>::isFullQueue() const
{
    return (count == maxQueueSize);
} //end isFullQueue
```

# Initialize Queue

- Initializes queue to empty state
  - First element added at the first array position
  - Initialize `queueFront` to zero, `queueRear` to `maxQueueSize` - one, `count` to zero

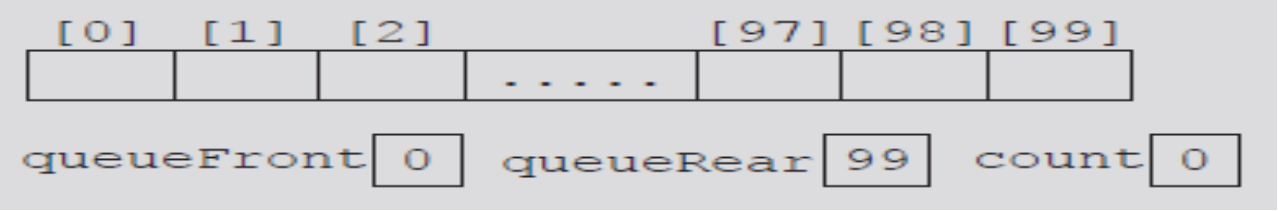



**FIGURE 8-10** Empty queue

```
template <class Type>
void queueType<Type>::initializeQueue()
{
    queueFront = 0;
    queueRear = maxQueueSize - 1;
    count = 0;
} //end initializeQueue
```

Why?

# Front

- Returns first queue element
  - If the queue nonempty
    - Element indicated by index `queueFront` returned
  - Otherwise
    - Program terminates

```
template <class Type>
Type queueType<Type>::front() const
{
    assert(!isEmptyQueue());
    return list[queueFront];
} //end front
```

# Back

- Returns last queue element
  - If queue nonempty
    - Returns element indicated by index `queueRear`
  - Otherwise
    - Program terminates

```
template <class Type>
Type queueType<Type>::back() const
{
    assert(!isEmptyQueue());
    return list[queueRear];
} //end back
```

# Add Queue

```
template <class Type>
void queueType<Type>::addQueue(const Type& newElement)
{
    if (!isFullQueue())
    {
        queueRear = (queueRear + 1) % maxQueueSize; //use the
                                //mod operator to advance queueRear
                                //because the array is circular
        count++;
        list[queueRear] = newElement;
    }
    else
        cout << "Cannot add to a full queue." << endl;
} //end addQueue
```

# Delete Queue

```
template <class Type>
void queueType<Type>::deleteQueue()
{
    if (!isEmptyQueue())
    {
        count--;
        queueFront = (queueFront + 1) % maxQueueSize; //use the
                     //mod operator to advance queueFront
                     //because the array is circular
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
} //end deleteQueue
```

# Constructors and Destructors

```
template <class Type>
queueType<Type>::queueType(int queueSize)
{
    if (queueSize <= 0)
    {
        cout << "Size of the array to hold the queue must "
             << "be positive." << endl;
        cout << "Creating an array of size 100." << endl;

        maxQueueSize = 100;
    }
    else
        maxQueueSize = queueSize;   //set maxQueueSize to
                                    //queueSize

    queueFront = 0;                 //initialize queueFront
    queueRear = maxQueueSize - 1;   //initialize queueRear
    count = 0;
    list = new Type[maxQueueSize];  //create the array to
                                    //hold the queue elements
} //end constructor
```

# Constructors and Destructors (cont'd.)

- Array storing queue elements
  - Created dynamically
  - When queue object goes out of scope
    - Destructor deallocates memory occupied by the array storing queue elements

```
template <class Type>
queueType<Type>::~queueType()
{
    delete [] list;
}
```

Exercise: Copy constructor and overloaded assignment operator

# Array-based Queue: Exercises

- Given an empty queue implemented using a static array of 5 elements, what will be the values of queueFront and queueRear, after the code is run? What does the queue look like?

```
ch = 'A'
for (int i=1; i<=4; i++)
{
   q.addQueue(ch);
   ch++;
   q.addQueue(ch);
   ch=q.front();
   q.deleteQueue();
}
```

# Array-based Queue: w/o count

- How to modify the class definition w/ the 2nd solution?
  - No member variable count

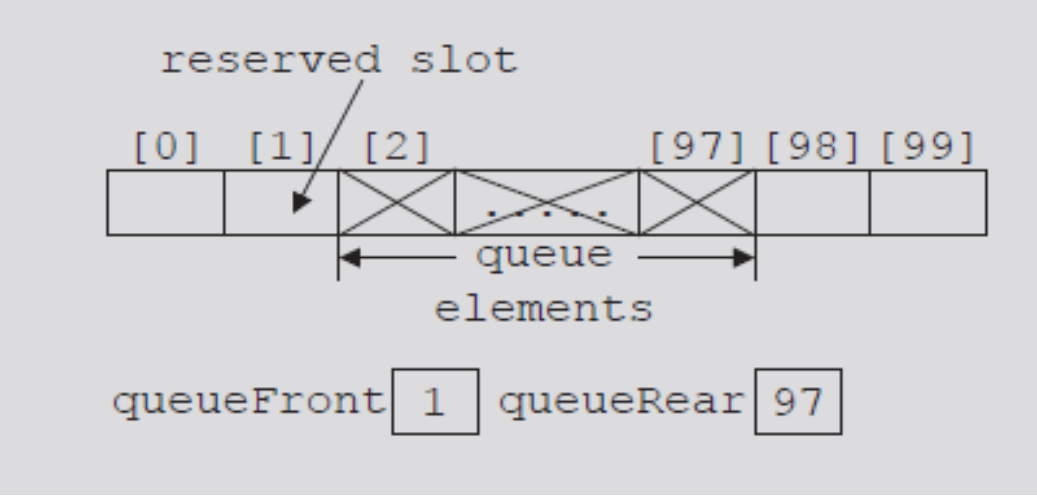



**FIGURE 8-9** Array to store the queue elements with a reserved slot

# Linked Implementation of Queues

- Array implementation issues
  - Fixed array size
    - Finite number of queue elements
  - Requires special array treatment with the values of the indices `queueFront`, `queueRear`
- Linked implementation of a queue
  - Simplifies special cases of the array implementation
  - Queue never full
- See code on pages 464-465

```
template <class Type>
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

template <class Type>
class linkedQueueType: public queueADT<Type>
{
public:
    const linkedQueueType<Type>& operator=(const linkedQueueType<Type>&);
    bool isEmptyQueue() const;
    bool isFullQueue() const;
    void initializeQueue();
    Type front() const;
    Type back() const;
    void addQueue(const Type& queueElement);
    void deleteQueue();
    linkedQueueType();
    linkedQueueType(const linkedQueueType<Type>& otherQueue);
    ~linkedQueueType();

private:
    nodeType<Type> *queueFront; //pointer to the front of the queue
    nodeType<Type> *queueRear;  //pointer to the rear of the queue
};
```

# Empty and Full Queue

- Empty queue if `queueFront` is `NULL`
- Memory allocated dynamically
  - Queue never full
  - Function implementing `isFullQueue` operation returns the value `false`

```
template <class Type>
bool linkedQueueType<Type>::isEmptyQueue() const
{
    return(queueFront == NULL);
} //end

template <class Type>
bool linkedQueueType<Type>::isFullQueue() const
{
    return false;
} //end isFullQueue
```

# Initialize Queue

- Initializes queue to an empty state
  - Empty if no elements in the queue

```
template <class Type>
void linkedQueueType<Type>::initializeQueue()
{
    nodeType<Type> *temp;

    while (queueFront!= NULL)  //while there are elements left
                               //in the queue
    {
        temp = queueFront;  //set temp to point to the current node
        queueFront = queueFront->link;  //advance first to
                                        //the next node
        delete temp;    //deallocate memory occupied by temp
    }

    queueRear = NULL;  //set rear to NULL
} //end initializeQueue
```

# `addQueue` Operation

- `addQueue` operation adds a new element at end of the queue
  - Access the pointer `queueRear`

```
template <class Type>
void linkedQueueType<Type>::addQueue(const Type& newElement)
{
   nodeType<Type> *newNode;
   newNode = new nodeType<Type>; //create the node
   newNode->info = newElement; //store the info
   newNode->link = NULL; //initialize the link field to NULL
   if (queueFront == NULL) //if initially the queue is empty
   {
      queueFront = newNode;
      queueRear = newNode;
   }
   else //add newNode at the end
   {
      queueRear->link = newNode;
      queueRear = queueRear->link;
   }
}//end addQueue
```

# `front` Operation

- If queue nonempty
  - Operation `front` returns first element
  - Element indicated `queueFront` returned
- If queue empty: `front` terminates the program

```
template <class Type>
Type linkedQueueType<Type>::front() const
{
    assert(queueFront != NULL);
    return queueFront->info;
} //end front
```

# `back` Operation

- If queue nonempty
  - Operation `back` returns last element
  - Element indicated by `queueRear` returned
- If queue empty: `back` terminates the program

```
template <class Type>
Type linkedQueueType<Type>::back() const
{
    assert(queueRear!= NULL);
    return queueRear->info;
} //end back
```

# deleteQueue Operation

- If queue nonempty
  - Operation deleteQueue removes first element
    - Access pointer queueFront

```
template <class Type>
void linkedQueueType<Type>::deleteQueue()
{
    nodeType<Type> *temp;

    if (!isEmptyQueue())
    {
        temp = queueFront;  //make temp point to the first node
        queueFront = queueFront->link; //advance queueFront

        delete temp;    //delete the first node

        if (queueFront == NULL) //if after deletion the
                                //queue is empty
            queueRear = NULL;   //set queueRear to NULL
    }
    else
        cout << "Cannot remove from an empty queue" << endl;
}//end deleteQueue
```

# Constructor & destructor

- Default constructor

```
template<class Type>
linkedQueueType<Type>::linkedQueueType()
{
    queueFront = NULL; //set front to null
    queueRear = NULL;  //set rear to null
} //end default constructor
```

Exercise: Copy constructor and overloaded assignment operator

- Destructor
  - deallocate memory used by elements
  - similar to function `initializeQueue`

# Queue Derived from the `class` `unorderedLinkedListType`

- Linked queue implementation
  - Similar to forward manner linked list implementation
  - Similar operations
    - `add Queue`, `insertFirst`
    - `initializeQueue` , `initializeList`
    - `isEmptyQueue`, `isEmptyList`
  - `deleteQueue` operation implemented as before
  - Same pointers
    - `queueFront` and `first`, `queueRear` and `last`

# Queue Derived from the `class unorderedLinkedListType` (cont'd.)

- Linked queue implementation (cont'd.)
  - Can derive the class to implement the queue from the `class linkedListType`
  - `class linkedListType`
    - An abstract class
    - Does not implement all operations
  - `class unorderedLinkedListType`
    - Derived from `class linkedListType`
    - Provides definitions of the abstract functions of the `class linkedListType`

# STL `class queue` (Queue Container Adapter)

- Standard Template Library (STL)
  - Provides a class to implement queues in a program
  - `Queue`
    - Name of class defining the queue
    - Name of header defining `class queue`

      `#include <queue>`
    - Provides relational operators comparing two queues
    - See Example 8-2

# STL `class queue` (cont'd.)

**TABLE 8-1** Operations on a `queue` object

addQueue

deleteQueue

| Operation | Effect |
|---|---|
| `size` | Returns the actual number of elements in the queue. |
| `empty` | Returns **`true`** if the queue is empty, and **`false`** otherwise. |
| `push(item)` | Inserts a copy of `item` into the queue. |
| `front` | Returns the next—that is, first—element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function. |
| `back` | Returns the last element in the queue, but does not remove the element from the queue. This operation is implemented as a value-returning function. |
| `pop` | Removes the next element in the queue. |

```
#include <iostream>                                    //Line 1
#include <queue>                                       //Line 2

using namespace std;                                   //Line 3

int main()                                             //Line 4
{                                                      //Line 5
    queue<int> intQueue;                               //Line 6

    intQueue.push(26);                                 //Line 7
    intQueue.push(18);                                 //Line 8
    intQueue.push(50);                                 //Line 9
    intQueue.push(33);                                 //Line 10

    cout << "Line 11: The front element of intQueue: "
         << intQueue.front() << endl;                  //Line 11

    cout << "Line 12: The last element of intQueue: "
         << intQueue.back() << endl;                   //Line 12

    intQueue.pop();                                    //Line 13

    cout << "Line 14: After the pop operation, the "
         << "front element of intQueue: "
         << intQueue.front() << endl;                  //Line 14

    cout << "Line 15: intQueue elements: ";            //Line 15

    while (!intQueue.empty())                          //Line 16
    {                                                  //Line 17
        cout << intQueue.front() << " ";               //Line 18
        intQueue.pop();                                //Line 19
    }                                                  //Line 20

    cout << endl;                                      //Line 21

    return 0;
}
```

```
Line 11: The front element of intQueue: 26
Line 12: The last element of intQueue: 33
Line 14: After the pop operation, the front element of intQueue: 18
Line 15: intQueue elements: 18 50 33
```

# Priority Queues

- Queue structure ensures items processed in the order received
- Priority queues
  - Customers (jobs) with higher priority pushed to the front of the queue
- Implementation
  - Ordinary linked list
    - Keeps items in order from the highest to lowest priority
  - Treelike structure
    - Very effective
    - Chapter 10

# STL `class priority_queue`

- `class` template `priority_queue<elemType>`
  - Queue element data type specified by `elemType`
  - Contained in the STL header file `queue`
- Specifying element priority
  - Default priority criteria for the queue elements
    - Less-than operator (<)
  - Overloading the less-than operator (<)
    - Compare the elements
  - Defining a comparison function to specify the priority

# Application of Queues: Simulation

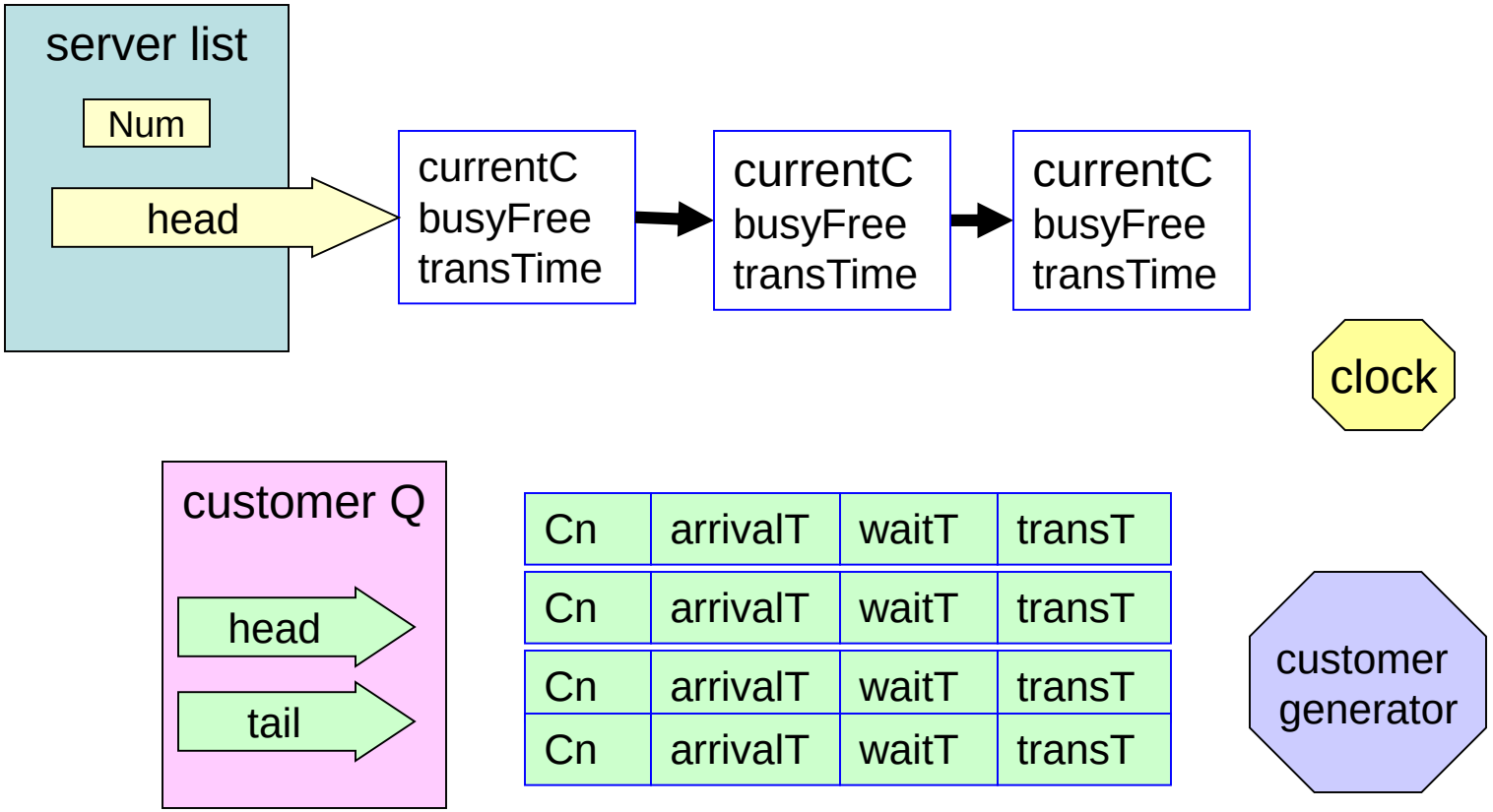
- Simulation
  - Technique in which one system models the behavior of another system
- Computer simulation
  - Represents objects being studied as data
  - Actions implemented with algorithms
    - Programming language implements algorithms with functions
    - Functions implement object actions
- Change in simulation results occurs if change in data value or modification of function definitions occurs
- Main goal
  - Generate results showing the performance of an existing system
  - Predict performance of a proposed system

# Designing a Queuing System

- Example: bank, movie theater, etc
- Server
  - Object that provides the service
- Customer
  - Object receiving the service
- Transaction time (service time)
  - Time required to serve a customer
- Queuing system consists of servers, queue of waiting objects
  - Model system consisting of a list of servers; waiting queue holding the customers to be served

# Designing a Queuing System (cont'd.)

- Modeling a queuing system: requirements
  - Number of servers, expected customer arrival time, time between customer arrivals, number of events affecting system
- Time-driven simulation
  - Clock implemented as a counter
  - Passage of time
    - Implemented by incrementing counter by one
- Run simulation for fixed amount of time
  - Example: run for 100 minutes
    - Counter starts at one and goes up to 100 using a loop


server list
Num
head
currentC
busyFree
transTime
currentC
busyFree
transTime
currentC
busyFree
transTime
clock
customer Q
head
tail
Cn
arrivalT
waitT
transT
Cn
arrivalT
waitT
transT
Cn
arrivalT
waitT
transT
Cn
arrivalT
waitT
transT
customer
generator

# Customer

- Has a customer number, arrival time, waiting time, transaction time, departure time
  - With known arrival time, waiting time, transaction time
    - Can determine departure time (add these three times)
- See `class customerType` code on pages 475-476
  - Implements customer as an ADT
- Member function definitions
  - Functions `setWaitingTime`, `getArrivalTime`, `getTransactionTime`, `getCustomerNumber`
    - Left as exercises

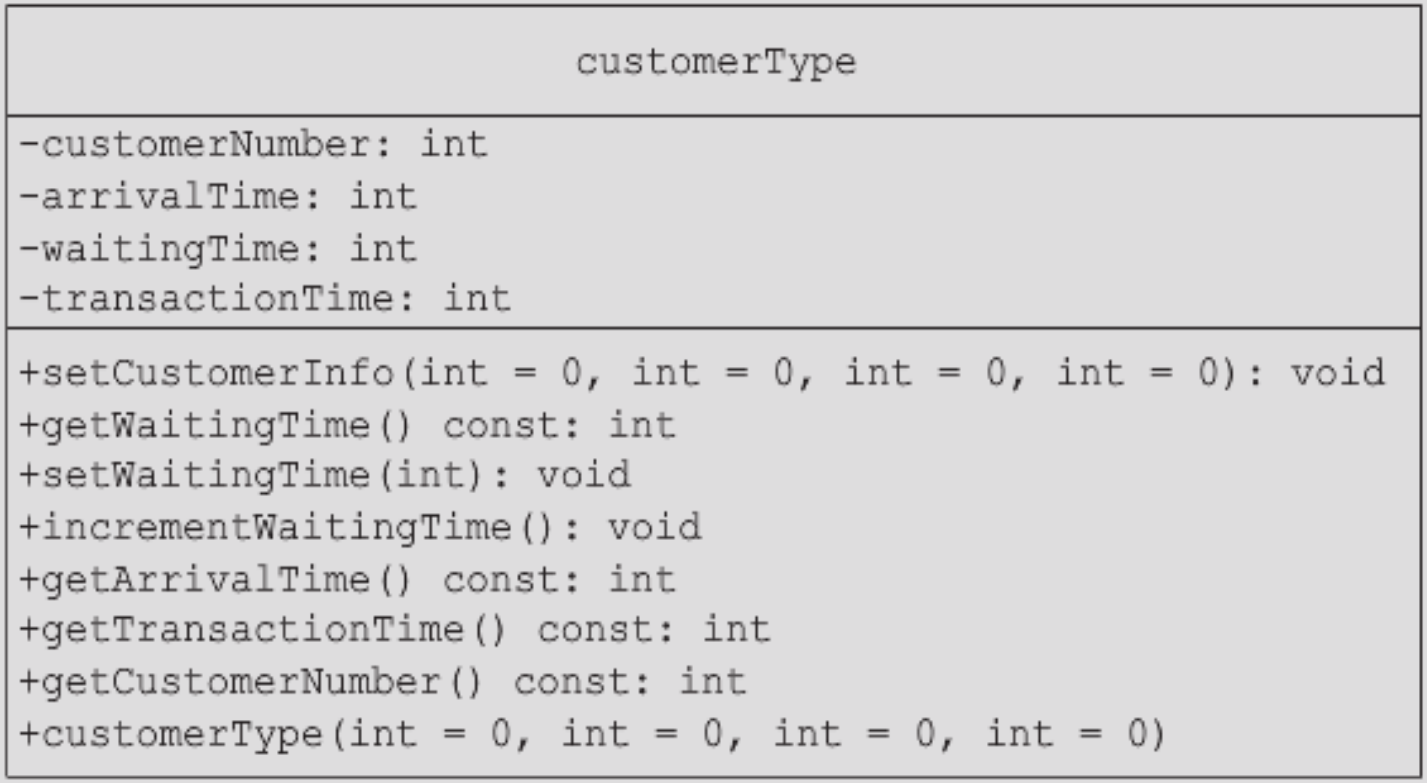



**FIGURE 8-11** UML diagram of the `class customerType`

```
class customerType
{
public:
    customerType(int cN = 0, int arrvTime = 0, int wTime = 0,
                 int tTime = 0);
      //Constructor to initialize the instance variables
      //according to the parameters
      //If no value is specified in the object declaration,
      //the default values are assigned.
      //Postcondition: customerNumber = cN; arrivalTime = arrvTime;
      //    waitingTime = wTime; transactionTime = tTime

    void setCustomerInfo(int cN = 0, int inTime = 0,
                         int wTime = 0, int tTime = 0);
      //Function to initialize the instance variables.
      //Instance variables are set according to the parameters.
      //Postcondition: customerNumber = cN; arrivalTime = arrvTime;
      //    waitingTime = wTime; transactionTime = tTime;

    int getWaitingTime() const;
      //Function to return the waiting time of a customer.
      //Postcondition: The value of waitingTime is returned.

    void setWaitingTime(int time);
      //Function to set the waiting time of a customer.
      //Postcondition: waitingTime = time;

    void incrementWaitingTime();
      //Function to increment the waiting time by one time unit.
      //Postcondition: waitingTime++;

    int getArrivalTime() const;
      //Function to return the arrival time of a customer.
      //Postcondition: The value of arrivalTime is returned.

    int getTransactionTime() const;
      //Function to return the transaction time of a customer.
      //Postcondition: The value of transactionTime is returned.

    int getCustomerNumber() const;
      //Function to return the customer number.
      //Postcondition: The value of customerNumber is returned.

private:
    int customerNumber;
    int arrivalTime;
    int waitingTime;
    int transactionTime;
};
```

```
void customerType::setCustomerInfo(int cN, int arrvTime,
                                   int wTime, int tTime)
{
    customerNumber = cN;
    arrivalTime = arrvTime;
    waitingTime = wTime;
    transactionTime = tTime;
}

customerType::customerType(int cN, int arrvTime,
                           int wTime, int tTime)
{
    setCustomerInfo(cN, arrvTime, wTime, tTime);
}

int customerType::getWaitingTime() const
{
    return waitingTime;
}


void customerType::incrementWaitingTime()
{
     waitingTime++;
}
```

# Server

- At any given time unit
  - Server either busy serving a customer or free
- String variable sets server status
- Every server has a timer
- Program might need to know which customer served by which server
  - Server stores information of the customer being served
- Three member variables associated with a server
  - `status`, `transactionTime`, `currentCustomer`

# Server (cont'd.)

- Basic operations performed on a server
  - Check if server free
  - Set server as free
  - Set server as busy
  - Set transaction time
  - Return remaining transaction time
  - If server busy after each time unit
    - Decrement transaction time by one time unit
- See `class serverType` code on page 477
  - Implements server as an ADT
- Member function definitions

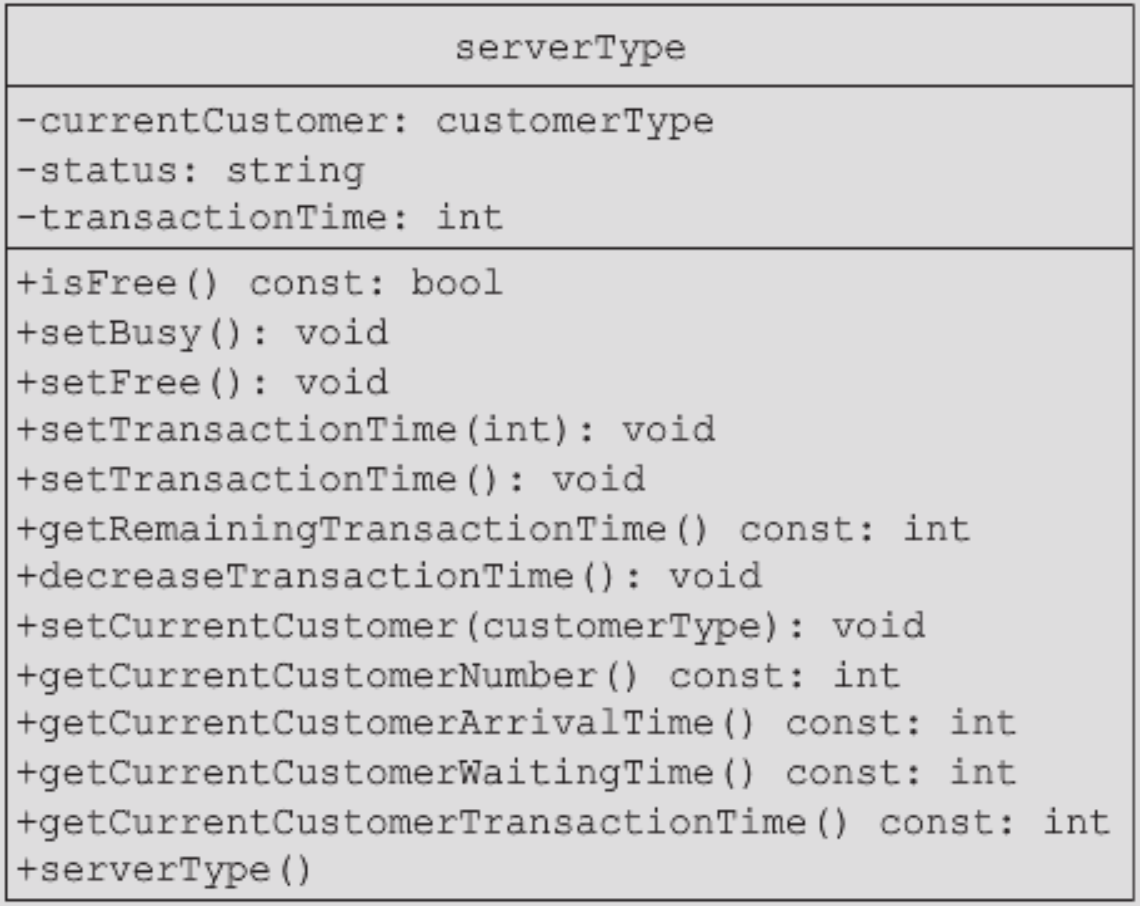



**FIGURE 8-12** UML diagram of the `class serverType`

```
class serverType
{
public:
    serverType();
      //Default constructor
      //Sets the values of the instance variables to their default
      //values.
      //Postcondition: currentCustomer is initialized by its
      //    default constructor; status = "free"; and the
      //    transaction time is initialized to 0.

    bool isFree() const;
      //Function to determine if the server is free.
      //Postcondition: Returns true if the server is free,
      //    otherwise returns false.

    void setBusy();
      //Function to set the status of the server to busy.
      //Postcondition: status = "busy";

    void setFree();
      //Function to set the status of the server to "free."
      //Postcondition: status = "free";

    void setTransactionTime(int t);
      //Function to set the transaction time according to the
      //parameter t.
      //Postcondition: transactionTime = t;

    void setTransactionTime();
      //Function to set the transaction time according to
      //the transaction time of the current customer.
      //Postcondition:
      //   transactionTime = currentCustomer.transactionTime;

    int getRemainingTransactionTime() const;
      //Function to return the remaining transaction time.
      //Postcondition: The value of transactionTime is returned.

    void decreaseTransactionTime();
      //Function to decrease the transactionTime by 1 unit.
      //Postcondition: transactionTime--;

    void setCurrentCustomer(customerType cCustomer);
      //Function to set the info of the current customer
      //according to the parameter cCustomer.
      //Postcondition: currentCustomer = cCustomer;

    int getCurrentCustomerNumber() const;
      //Function to return the customer number of the current
      //customer.
      //Postcondition: The value of customerNumber of the
      //    current customer is returned.

    int getCurrentCustomerArrivalTime() const;
      //Function to return the arrival time of the current
      //customer.
      //Postcondition: The value of arrivalTime of the current
      //    customer is returned.

    int getCurrentCustomerWaitingTime() const;
      //Function to return the current waiting time of the
      //current customer.
      //Postcondition: The value of transactionTime is returned.

    int getCurrentCustomerTransactionTime() const;
      //Function to return the transaction time of the
      //current customer.
      //Postcondition: The value of transactionTime of the
      //    current customer is returned.

private:
    customerType currentCustomer;
    string status;
    int transactionTime;
};
```

```
serverType::serverType()
{
    status = "free";
    transactionTime = 0;
}

bool serverType::isFree() const
{
    return (status == "free");
}

void serverType::setBusy()
{
    status = "busy";
}

void serverType::setFree()
{
    status = "free";
}

void serverType::setTransactionTime(int t)
{
    transactionTime = t;
}

void serverType::setTransactionTime()
{
    int time;

    time = currentCustomer.getTransactionTime();

    transactionTime = time;
}

void serverType::decreaseTransactionTime()
{
    transactionTime--;
}
```

# Server List

- Set of servers
- `class serverListType`
  - Two member variables
    - Store number of servers
    - Maintain a list of servers
  - List of servers created during program execution
  - Several operations must be performed on a server list
  - See `class serverListType` code on page 481
    - Implements the list of servers as an ADT
  - Definitions of member functions

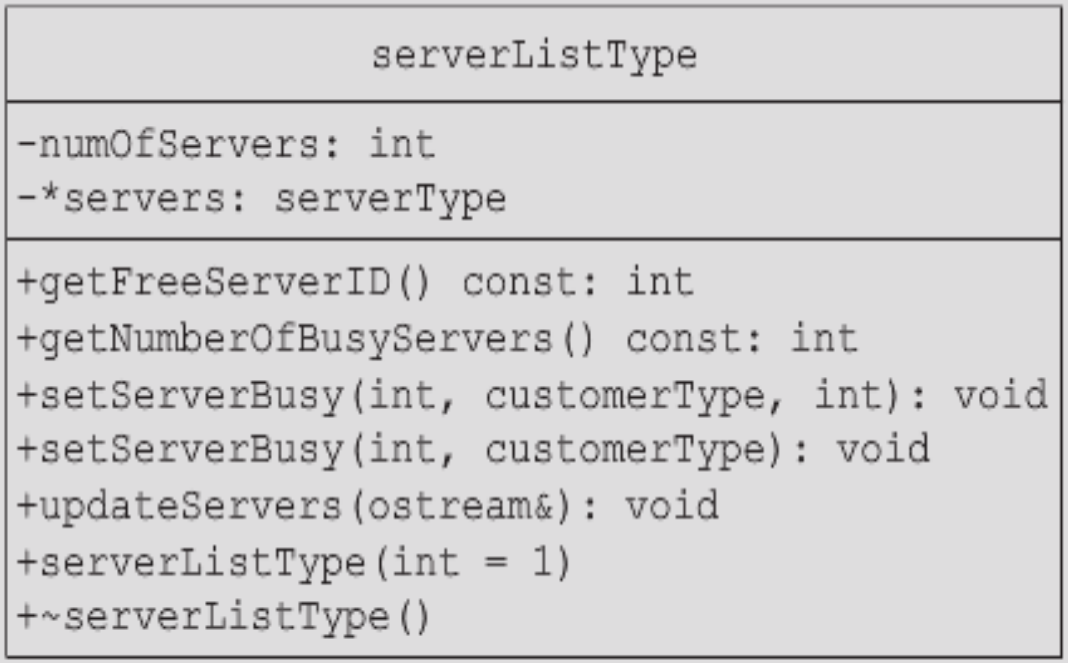



**FIGURE 8-13** UML diagram of the `class serverListType`

```
class serverListType
{
public:
    serverListType(int num = 1);
      //Constructor to initialize a list of servers
      //Postcondition: numOfServers = num
      //    A list of servers, specified by num, is created and
      //    each server is initialized to "free".

    ~serverListType();
      //Destructor
      //Postcondition: The list of servers is destroyed.

    int getFreeServerID() const;
      //Function to search the list of servers.
      //Postcondition: If a free server is found, returns its ID;
      //    otherwise, returns -1.

    int getNumberOfBusyServers() const;
      //Function to return the number of busy servers.
      //Postcondition: The number of busy servers is returned.

    void setServerBusy(int serverID, customerType cCustomer,
                       int tTime);
      //Function to set a server busy.
      //Postcondition: The server specified by serverID is set to
      //    "busy", to serve the customer specified by cCustomer,
      //    and the transaction time is set according to the
      //    parameter tTime.

    void setServerBusy(int serverID, customerType cCustomer);
      //Function to set a server busy.
      //Postcondition: The server specified by serverID is set to
      //    "busy", to serve the customer specified by cCustomer.

    void updateServers(ostream& outFile);
      //Function to update the status of a server.
      //Postcondition: The transaction time of each busy server
      //    is decremented by one unit. If the transaction time of
      //    a busy server is reduced to zero, the server is set to
      //    "free". Moreover, if the actual parameter corresponding
      //    to outFile is cout, a message indicating which customer
      //    has been served is printed on the screen, together with the
      //    customer's departing time. Otherwise, the output is sent
      //    to a file specified by the user.

private:
    int numOfServers;
    serverType *servers;
};
```

```
serverListType::serverListType(int num)
{
    numOfServers = num;
    servers = new serverType[num];
}

serverListType::~serverListType()
{
    delete [] servers;
}

int serverListType::getFreeServerID() const
{
    int serverID = -1;

    for (int i = 0; i < numOfServers; i++)
        if (servers[i].isFree())
        {
            serverID = i;
            break;
        }

    return serverID;
}

int serverListType::getNumberOfBusyServers() const
{
    int busyServers = 0;

    for (int i = 0; i < numOfServers; i++)
        if (!servers[i].isFree())
            busyServers++;

    return busyServers;
}
```

```
void serverListType::setServerBusy(int serverID,
                                   customerType cCustomer, int tTime)
{
    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(tTime);
    servers[serverID].setCurrentCustomer(cCustomer);
}
void serverListType::setServerBusy(int serverID,
                                        customerType cCustomer)
{
    int time = cCustomer.getTransactionTime();

    servers[serverID].setBusy();
    servers[serverID].setTransactionTime(time);
    servers[serverID].setCurrentCustomer(cCustomer);
}
void serverListType::updateServers(ostream& outF)
{
   for (int i = 0; i < numOfServers; i++)
       if (!servers[i].isFree())
       {
          servers[i].decreaseTransactionTime();

          if (servers[i].getRemainingTransactionTime() == 0)
          {
              outF << "From server number " << (i + 1)
                   << " customer number "
                   << servers[i].getCurrentCustomerNumber()
                   << "\n     departed at clock unit "
                   << servers[i].getCurrentCustomerArrivalTime()
                   + servers[i].getCurrentCustomerWaitingTime()
                   + servers[i].getCurrentCustomerTransactionTime()
                   << endl;
              servers[i].setFree();
          }
       }
}
```

# Waiting Customers Queue

- Upon arrival, customer goes to end of queue
  - When server available
    - Customer at front of queue leaves to conduct transaction
  - After each time unit, waiting time incremented by one
- Derive `class waitingCustomerQueueType` from `class queueType`
  - Add additional operations to implement the customer queue
  - See code on page 485

```
class waitingCustomerQueueType: public queueType<customerType>
{
public:
    waitingCustomerQueueType(int size = 100);
      //Constructor
      //Postcondition: The queue is initialized according to the
      //    parameter size. The value of size is passed to the
      //    constructor of queueType.

    void updateWaitingQueue();
      //Function to increment the waiting time of each
      //customer in the queue by one time unit.
};


waitingCustomerQueueType::waitingCustomerQueueType(int size)
                          :queueType<customerType>(size)
{
}

void waitingCustomerQueueType::updateWaitingQueue()
{
    customerType cust;

    cust.setWaitingTime(-1);
    int wTime = 0;

    addQueue(cust);

    while (wTime != -1)
    {
        cust = front();
        deleteQueue();

        wTime = cust.getWaitingTime();
        if (wTime == -1)
            break;
        cust.incrementWaitingTime();
        addQueue(cust);
    }
}
```

# Main Program

- Run the simulation
  - Need information (simulation parameters)
    - Number of time units the simulation should run
    - The number of servers
    - Transaction time
    - Approximate time between customer arrivals
  - Function `setSimulationParameters`
    - Prompts user for these values
    - See code on page 487

# Main Program (cont'd.)

- General algorithm to start the transaction

1. Remove the customer from the front of the queue.

```
customer = customerQueue.front();
customerQueue.deleteQueue();
```

2. Update the total waiting time by adding the current customer's waiting time to the previous total waiting time.

```
totalWait = totalWait + customer.getWaitingTime();
```

3. Set the free server to begin the transaction.

```
serverList.setServerBusy(serverID, customer, transTime);
```

```
queue<customerType> customerQueue;
serverListType serverList(numberOfServers);

for (clock = 1; clock <= simulationTime; clock++)
{
    serverList.updateServers(cout);
    if (!customerQueue.empty()) updateCustQueue(customerQueue);

    if (isCustomerArrived(timeBetweenCustomerArrival))
    {
        custNumber++;
        customer.setCustomerInfo(custNumber,clock,0, transactionTime);
        customerQueue.push(customer);
    }

    serverID = serverList.getFreeServerID();
    if (serverID != -1 && !customerQueue.empty())
     {
        customer = customerQueue.front();
        customerQueue.pop();
        serverList.setServerBusy(serverID, customer);
    }
}
```

set up data structure

check server, check cQ

add new arrival

if server free, assign new customer

# Main Program (cont'd.)

- Use the Poisson distribution from statistics
  - Probability of *y* events occurring at a given time
    - Where λ is the expected value that y events occur at that time

$$P(y) = \frac{\lambda^y e^{-\lambda}}{y!}, y = 0, 1, 2, \ldots,$$

  - $P(0) = e^{-\lambda}$ : probability that no one arrives
  - $1 - P(0) = 1 - e^{-\lambda}$ : probability that some customers arrive
  - We approximate $1 - P(0)$ to be probability of *exactly one* customer arrive, since multiple customers arrival probability is negligible
- Function `runSimulation` implements the simulation
  - Function `main` is simple and straightforward
    - Calls only the function `runSimulation`

```
#include <iostream>
#include <cstdlib>
#include <iomanip>
#include <cmath>

bool isCustomerArrived(double arvTimeDiff)
{
    double value;

    value = static_cast<double> (rand()) / static_cast<double>(RAND_MAX);

    return (value > exp(- 1.0/arvTimeDiff));
}
```

# Summary

- Queue
  - First In First Out (FIFO) data structure
  - Implemented as array or linked list
  - Linked lists: queue never full
- Standard Template Library (STL)
  - Provides a class to implement a queue in a program
- Priority Queue
  - Customers with higher priority pushed to the front
- Simulation
  - Common application for queues

# Self Exercises

- Programming Exercises: 1, 2, 4, 6, 7