

Ch 8

Queues

- Introduction to Queues and the STL Queue
- Queue Applications
- Implementations of the Queue Class
- Priority Queues
- Reference Return Values for the Stack, Queue and Priority Queue Classes

Introduction to Queues and the STL Queue Class

Basic Definitions and Operations

Queue Definitions

- A queue is a data type (not a data structure) consisting of ordered data items such that they may be inserted at one end (called the rear) and removed from the other end (called the front). The entry at the front end is called the first entry.
- Queues obey first-in/first-out (FIFO) rules. Data items are removed from a queue in the same order of their insertion.
- Queues, therefore, are containers or list data types with specific additional properties.

The queue STL Class

The stack class is formally declared as:

```
template <class T, class Container = deque<T>>  
Class queue { /* ... */ };
```

Its constructor has the form:

```
explicit queue(const Container& cnt = Container());
```

Mutator functions:

```
void pop();  
void (push(const T& val);
```

Accessor functions:

```
bool empty() const;  
size_type& size() const;  
value_type& front();  
const value_type& front() const;  
value_type& back();  
const value_type& back() const;
```

Example: Echoing a Word

while (there are more characters)

Read a character.

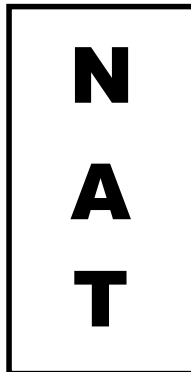
Push the character into the queue.

while (the queue is not empty)

Write out the front character.

Remove (pop) the front character.

Queue



Input



Output



Queue Applications

Recognizing Palindromes & Simulations

Example: Recognizing Palindromes

Straw? No, too stupid a fad. I put soot on warts.

```
while (there are more characters)
  Read a character.
  if (character is a letter)
    Push onto stack and queue.
mismatches = 0
while (the queue and stack are not empty)
  if (stack's top != queue's front)
    mismatches++
  Pop the stack and queue.
Return mismatches == 0.
```

pal.cxx

```
int main( ) {
    queue<char> q;
    stack<char> s;
    char letter;
    queue<char>::size_type mismatches = 0;

    cout << "Enter a line and I will see if it's a palindrome:" << endl;

    while (cin.peek( ) != '\n') {
        cin >> letter;
        if (isalpha(letter)) {
            q.push(toupper(letter));
            s.push(toupper(letter)); } }

    while ((!q.empty( )) && (!s.empty( ))) {
        if (q.front( ) != s.top( ))
            ++mismatches;
        q.pop( );
        s.pop( ); }

    if (mismatches == 0)
        cout << "That is a palindrome." << endl;
    else
        cout << "That is not a palindrome." << endl;
    return EXIT_SUCCESS;
}
```

Car Wash Simulation

```
for (current = 1; current <= length; current++)  
    if (bool_source == true)  
        Push current into queue.  
    if (washer not busy and queue not empty)  
        next = queue's front, pop queue.  
        Send (current - next) to the averager.  
        Send start message to washer.  
        Send a tick message to the washer.  
Display averager's average wait time and number  
served.
```

washing.h...

```
class bool_source {  
  public:  
    bool_source(double p = 0.5);  
    bool query( ) const;  
  private:  
    double probability;  
};  
  
class averager {  
  public:  
    averager( );  
    void next(double value);  
    std::size_t count( ) const {  
      return count; }  
    double average( ) const;  
  private:  
    std::size_t count;  
    double sum;  
};
```

...washing.h

```
class washer {  
  public:  
    washer(unsigned int s = 60);  
    void tick( );  
    void start( );  
    bool is_busy( ) const {  
      return (wash_time_left > 0); }  
  private:  
    unsigned int seconds_for_wash;  
    unsigned int wash_time_left;  
};
```

washing.cxx...

```
bool_source::bool_source(double p) {
    assert(p >= 0);
    assert(p <= 1);
    probability = p;
}

bool bool_source::query( ) const {
    return (rand( ) < probability * RAND_MAX);
}

averager::averager( ) {
    count = 0;
    sum = 0;
}

void averager::next(double value) {
    ++count;
    sum += value;
}

double averager::average( ) const {
    assert(count( ) > 0);
    return sum / count;
}
```

...washing.cxx

```
washer::washer(unsigned int s) {
    seconds_for_wash = s;
    wash_time_left = 0;
}

void washer::tick( ) {
    if (is_busy( ))
        --wash_time_left;
}

void washer::start( ) {
    assert(!is_busy( ));
    wash_time_left = seconds_for_wash;
}
```

carwash.cxx...

```
void car_wash_simulate
(unsigned int wash_time,
 double arrival_prob,
 unsigned int total_time);

int main( ) {
    car_wash_simulate(240, 1.0 / 400, 6000);
    return EXIT_SUCCESS;
}
```

...carwash.cxx...

```
void car_wash_simulate
(unsigned int wash_time,
 double arrival_prob,
 unsigned int total_time) {
    queue<unsigned int> arrival_times;
    unsigned int next;
    bool_source arrival(arrival_prob);
    washer machine(wash_time);
    averager wait_times;
    unsigned int curSec;

    cout << "Seconds to wash one car: " << wash_time << endl;
    cout << "Probability of customer arrival during a second: ";
    cout << arrival_prob << endl;
    cout << "Total simulation seconds: " << total_time << endl;
```

...carwash.cxx

```
for (curSec = 1; curSec <= total_time; ++curSec) {
    if (arrival.query( ))
        arrival_times.push(curSec);
    if ((!machine.is_busy( )) && (!arrival_times.empty( ))) {
        next = arrival_times.front( );
        arrival_times.pop( );
        wait_times.next(curSec - next);
        machine.start( ); }
    machine.tick( ); }

cout << "Customers served: "
     << wait_times.count( )
     << endl;
if (wait_times.count( ) > 0)
    cout << "Average wait: "
         << wait_times.average( )
         << " sec"
         << endl;
}
```

Approximating the Bell Curve

$$N:0,1 \approx \sum_{i=1}^{12} R_i - 6$$

A Normal Distribution Source

```
class NormalSource {  
  public:  
    NormalSource(bool r = true, double m = 0.0, s = 1.0);  
    double query() const;  
  private:  
    double mean;  
    double stdDev;  
};  
  
NormalSource::NormalSource(bool r, double m, double s) {  
  if (r)  
    srand(time(NULL));  
  mean = m;  
  stdDev = s;  
}  
  
double NormalSource::query() const {  
  double sum = 0.0;  
  for (int i = 0; i < 12; i++)  
    sum += (double)rand()/RAND_MAX;  
  return stdDev*(sum - 6) + mean;  
}
```

Project G

Complete programming project 9 from Chapter 8 of the text. Directions are on page 429.

- Place all **class** definitions in a single interface file and all implementations in a single file as well.
- Name the interface file **AirSim.h**.
- Name the implementation file **AirSim.cpp**.
- The client code should be modeled after the example for the car wash simulation, **carwash.cxx**. Name its file **AirportSimulation.cpp**. A template for that file can be found [here](#). Be sure to use it as a starting point.

When you're satisfied with the program, submit your work via the [CATE form for Project G](#).

Implementations of the Queue Class

Circular Arrays and Linked Lists

Queue Class Interface (Array Version)

```
template <class Item>
class queue {
public:
    typedef std::size_t size_type;
    typedef Item value_type;
    static const size_type CAPACITY = 30;

    queue( );

    void pop( );
    void push(const Item& entry);

    bool empty( ) const {
        return (count == 0); }
    Item front( ) const;
    size_type size( ) const {
        return count; }
private:
    Item data[CAPACITY];
    size_type first;
    size_type last;
    size_type count;
    size_type next_index(size_type i) const {
        return (i + 1) % CAPACITY; }
};
```

Queue Class Invariant (Array Version)

- The number of items in the queue is in **count**.
- For a non-empty queue, the items are stored in a circular array beginning at **data[first]** and continuing through **data[last]**. The capacity of the array is **CAPACITY**.
- For an empty queue, **last** is some valid index, and **first** is always equal to **next_index(last)**.

Queue Class Implementation (Array Version)...

```
template <class Item>
const typename queue<Item>::size_type
    queue<Item>::CAPACITY;

template <class Item>
queue<Item>::queue( ) {
    count = 0;
    first = 0;
    last = CAPACITY - 1;
}

template <class Item>
Item queue<Item>::front( ) const {
    assert(!empty( ));
    return data[first];
}
```

...Queue Class Implementation (Array Version)

```
template <class Item>
void queue<Item>::pop( ) {
    assert(!empty( ));
    first = next_index(first);
    --count;
}

template <class Item>
void queue<Item>::push(const Item& entry) {
    assert(size( ) < CAPACITY);
    last = next_index(last);
    data[last] = entry;
    ++count;
}
```

Queue Class Interface (List Version)

```
template <class Item>
class queue {
public:
    typedef std::size_t size_type;
    typedef Item value_type;

    queue( );

    queue(const queue<Item>& source);
    ~queue( );
    void operator =(const queue<Item>& source);

    void pop( );
    void push(const Item& entry);

    bool empty( ) const {
        return (count == 0); }
    Item front( ) const;
    size_type size( ) const {
        return count; }
private:
    node<Item> *front_ptr;
    node<Item> *rear_ptr;
    size_type count;
};
```

Queue Class Invariant (List Version)

- The number of items in the queue is stored in the member variable **count**.
- The items in the queue are stored in a linked list, with the front of the queue stored at the head node, and the rear of the queue stored at the final node.
- The member variable **front_ptr** is the head pointer of the linked list of items. For a non-empty queue, the member variable **rear_ptr** is the tail pointer of the linked list; for an empty list, **rear_ptr** is immaterial.

Queue Class Implementation (List Version)...

```
template <class Item>
queue<Item>::queue( ) {
    count = 0;
    front_ptr = NULL;
}

template <class Item>
queue<Item>::queue(const queue<Item>& source) {
    count = source.count;
    list_copy(source.front_ptr, front_ptr, rear_ptr);
}

template <class Item>
queue<Item>::~~queue( ) {
    list_clear(front_ptr);
}

template <class Item>
void queue<Item>::operator =(const queue<Item>& source) {
    if (this == &source)
        return ;
    list_clear(front_ptr);
    list_copy(source.front_ptr, front_ptr, rear_ptr);
    count = source.count;
}
```

...Queue Class Implementation (List Version)

```
template <class Item>
Item queue<Item>::front( ) const {
    assert(!empty( ));
    return front_ptr->data( );
}

template <class Item>
void queue<Item>::pop( ) {
    assert(!empty( ));
    list_head_remove(front_ptr);
    --count;
}

template <class Item>
void queue<Item>::push(const Item& entry) {
    if (empty( )) {
        list_head_insert(front_ptr, entry);
        rear_ptr = front_ptr; }
    else {
        list_insert(rear_ptr, entry);
        rear_ptr = rear_ptr->link( ); }
    ++count;
}
```