the Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
1. The Queue Abstract Data Type
   - queue ADT
     - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
A stack is a FIFO (First In First Out) sequence:
- pop: remove the first element,
- push: append last element.

Main applications:
- fair protocol to share resources,
- run simulations.

A queue can be linear or circular.
Applications for circular queues:
- waiting room with fixed #seats,
- load balancing, round-robin scheduling.
queue ADT

abstract <typename T> queue;
/* a queue is a sequence of elements, 
    FIFO = First In First Out */

abstract bool empty ( queue q );
postcondition: empty(s)
    == true if queue is empty,
    == false otherwise;

abstract T pop ( queue q );
precondition: not empty(q);
postcondition: first element is removed from q;

abstract void push ( queue q, elements e );
postcondition: element e is at end of queue q;
#include <queue>

The queue is a templated class, we can specify the type $T$ of the elements of the queue at instantiation, e.g.:

```cpp
queue<string> q;
```

declares a queue of strings.

Operations on $q$ for $t$ of type $T$:

- `q.push(t)` : appends $t$ to the end of $q$
- $t = q.front()$ : returns $t$ at front of $q$
- `q.pop()` : removes first element of $q$
- `q.empty()` : true if $q$ is empty
- `q.size()` : returns number of elements of $q$
the Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
using the STL queue

```cpp
#include <iostream>
#include <string>
#include <queue>
using namespace std;

int main()
{
    queue<string> q;
    cout << "pushing A, B, and C ..." << endl;
    q.push("A"); q.push("B"); q.push("C");
    cout << "size of queue : " << q.size() << endl;
    cout << "front of queue : " << q.front() << endl;
}
```

cout << "popping front ..." << endl;
q.pop();

for( ; !q.empty(); q.pop())
    cout << "front of queue : "
    << q.front() << endl;

if(q.empty())
    cout << "queue is empty" << endl;
else
    cout << "queue is not empty" << endl;
the Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
Simulating a Printer Queue

Consider a sequence of printer jobs.

Assume

- every job takes between 1 and 10 pages;
- arrival time between jobs is in range 10..20 minutes.

For simplicity, we use uniform distributions.

If printer has a capacity of 100 pages, how many times we need to reload for 100 jobs?
Our Simulation Algorithm

Two major subroutines:

1. generate $n$ jobs,
2. process the jobs.

The user is prompted to provide $n$.

Parameters are set with \#define.

For each job we store two integers:

- size: number of pages to print,
- arrival: elapsed time in minutes.

We use a struct to define a job.
the Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <queue>
using namespace std;

// parameters for the simulation
// 1. size of the jobs in [1,10]
define min_size 1
#define max_size 10
// 2. arrival time between jobs
#define min_time 10
#define max_time 20
// 3. capacity of printer
#define capacity 100
type and function declarations

struct Job
{
    int arrival; // arrival time in minutes
    int size;   // size of job in pages
};

int random_number ( int a, int b );
// returns a random integer number,
// uniformly distributed between a and b

queue<Job> generate_jobs ( int n );
// returns a queue of n jobs

void process_jobs ( queue<Job> q );
// runs the simulation, processing jobs in q
the main program

```c++
int main()
{
    int n;

    cout << "Give number of jobs : "; cin >> n;

    srand(time(0));
    queue<Job> q = generate_jobs(n);
    process_jobs(q);

    return 0;
}

int random_number ( int a, int b )
{
    int r = rand() % (b-a+1);
    return a + r;
}
```
generating jobs

```cpp
queue<Job> generate_jobs ( int n )
{
    queue<Job> q;
    int elapsed = 0;
    for(int i=0; i<n; i++)
    {
        Job J;
        J.size = random_number(min_size,max_size);
        cout << "job " << i << " has size " << J.size;
        J.arrival = elapsed
            + random_number(min_time,max_time);
        cout << " arrived at time "
            << J.arrival << endl;
        elapsed = J.arrival;
        q.push(J);
    }
    return q;
}
```
void process_jobs ( queue<Job> q )
{
    int printed = 0;
    int packs = 0;

    for(int i=0; !q.empty(); q.pop(), i++)
    {
        Job J = q.front();
        cout << "job " << i
            << " arrived at " << J.arrival
            << " has size " << J.size << endl;
        printed = printed + J.size;
    }
if (printed > capacity)
{
    cout << "please provide paper ...\n";
    cout << "continue ? (y/n) ";
    char c; cin >> c;
    if (c != 'y') break;

    printed = printed - capacity;
    packs = packs + 1;
}

cout << "printed "
    << packs * capacity + printed
    << " pages" << endl;
collecting statistics

With `srand(time(0))`, the answer will differ at each run.

Running \( N \) times, answers \( a_k, k = 1, 2, \ldots, N \), compute

- average \( \bar{a} = \frac{1}{N} \sum_{k=1}^{N} a_k \);
- standard deviation:

\[
d = \sqrt{\frac{1}{N} \sum_{k=1}^{N} (a_k - \mu)^2}.
\]

Law of large numbers, for \( N \to \infty \): \( \bar{a} \to \mu \) and \( d \to \sigma \), where \( \mu \) and \( \sigma \) are the true average and standard deviation for the answer.
the Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
We map list operations to queue operations.

For any $t$ of type $T$:

<table>
<thead>
<tr>
<th>queue$&lt;$T$&gt;$ q</th>
<th>list$&lt;$T$&gt;$ L</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.push($t$)</td>
<td>L.push_back($t$)</td>
</tr>
<tr>
<td>q.pop()</td>
<td>L.pop_front()</td>
</tr>
<tr>
<td>$t$ = q.front()</td>
<td>$t$ = L.front()</td>
</tr>
<tr>
<td>q.empty()</td>
<td>L.empty()</td>
</tr>
<tr>
<td>q.size()</td>
<td>L.size()</td>
</tr>
</tbody>
</table>
use STL list as queue

```cpp
#include <iostream>
#include <string>
#include <list>
using namespace std;

int main()
{
    list<string> q;
    cout << "pushing A, B, and C ..." << endl;
    q.push_back("A"); q.push_back("B");
    q.push_back("C");
    cout << "size of queue : " << q.size() << endl;
    cout << "front of queue : "
        << q.front() << endl;
}
```

cout << "popping front ..." << endl;
q.pop_front();

for( ; !q.empty(); q.pop_front())
    cout << "front of queue : 
    " << q.front() << endl;

if(q.empty())
    cout << "queue is empty" << endl;
else
    cout << "queue is not empty" << endl;
The Queue

1. The Queue Abstract Data Type
   - queue ADT
   - using the STL queue

2. Simulating a Printer Queue
   - designing the simulation
   - simulation with STL queue

3. adapting STL list and vector
   - using STL list as queue
   - using STL vector as queue
We map vector operations to queue operations.

For any $t$ of type $T$:

<table>
<thead>
<tr>
<th>queue$&lt;$T$&gt;$ q</th>
<th>vector$&lt;$T$&gt;$ v</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.push($t$)</td>
<td>v.push_back($t$)</td>
</tr>
<tr>
<td>q.pop()</td>
<td>v.erase(v.begin())</td>
</tr>
<tr>
<td>$t = q.front()$</td>
<td>$t = v.front()$</td>
</tr>
<tr>
<td>q.empty()</td>
<td>v.empty()</td>
</tr>
<tr>
<td>q.size()</td>
<td>v.size()</td>
</tr>
</tbody>
</table>
use STL vector as queue

```cpp
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main()
{
    vector<string> q;

    cout << "pushing A, B, and C ..." << endl;
    q.push_back("A"); q.push_back("B");
    q.push_back("C");
    cout << "size of queue : " << q.size() << endl;
    cout << "front of queue : "
        << q.front() << endl;
}```
cout << "poping front ..." << endl;

q.erase(q.begin());

for( ; !q.empty(); q.erase(q.begin()))
    cout << "front of queue : "
    << q.front() << endl;

if(q.empty())
    cout << "queue is empty" << endl;
else
    cout << "queue is not empty" << endl;
Started Chapter 6 on *Queues*, covered ADT and STL, illustrated with a basic simulation program.

**Exercises:**

1. Use a stack to reverse the order of the elements in a queue of strings. Write an interactive program to test your reversal function.

2. Combine the use of stack and queue to test whether a string is a palindrome.

3. Extend the `printer_queue.cpp` simulation to do multiple runs (for convenience of the user, toggle off the “pause” statement) and compute the average and standard deviation of the runs.