
12.2. Queues

The class `queue<>` implements a queue (also known as FIFO). With `push()`, you can insert any number of elements (Figure 12.3). With `pop()`, you can remove the elements in the same order in which they were inserted ("first in, first out"). Thus, a queue serves as a classic data buffer.

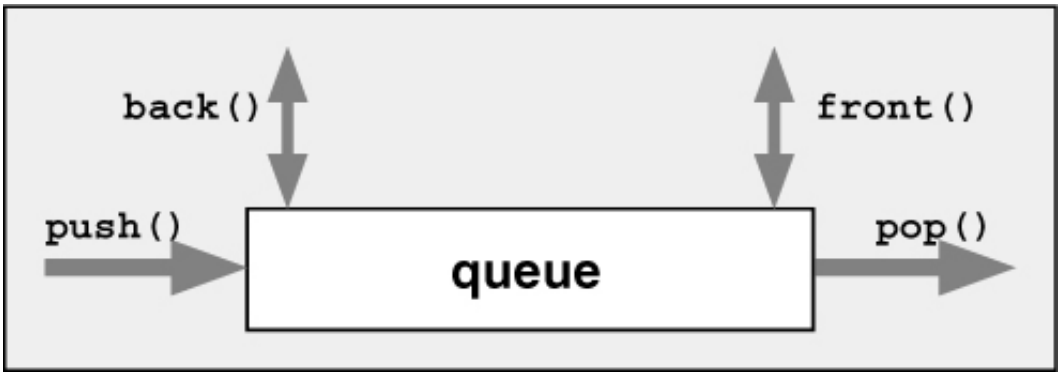

Figure 12.3. Interface of a Queue

To use a queue, you must include the header file `<queue>`:

```
#include <queue>
```

In `<queue>`, the class `queue` is defined as follows:

Click here to view code image

```
namespace std {
    template <typename T,
              typename Container = deque<T>>
             class  queue;
}
```

The first template parameter is the type of the elements. The optional second template parameter defines the container that the queue uses internally for its elements. The default container is a deque. For example, the following declaration defines a queue of strings:

```
std::queue<std::string> buffer;          // string queue
```

The queue implementation simply maps the operations into appropriate calls of the container that is used internally (Figure 12.4). You can use any sequence container class that provides the member functions `front()`, `back()`, `push_back()`, and `pop_front()`. For example, you could also use a list as the container for the elements:

```
std::queue<std::string,std::list<std::string>> buffer;
```

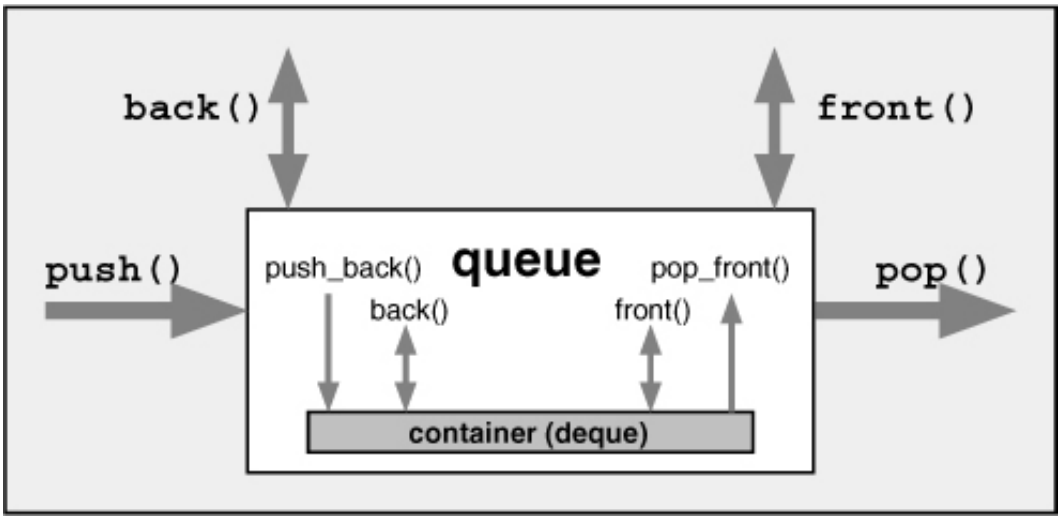

Figure 12.4. Internal Interface of a Queue

12.2.1. The Core Interface

The core interface of queues is provided by the member functions `push()`, `front()`, `back()` and `pop()`:

- **`push()`** inserts an element into the queue.
- **`front()`** returns the next available element in the queue (the element that was inserted first).
- **`back()`** returns the last available element in the queue (the element that was inserted last).
- **`pop()`** removes an element from the queue.

Note that `pop()` removes the next element but does not return it, whereas `front()` and `back()` return the element without removing it. Thus, you must always call `front()` and `pop()` to process and remove the next element from the queue. This interface is somewhat inconvenient, but it performs better if you want to only remove the next element without processing it. Note that the behavior of `front()`, `back()`, and `pop()` is undefined if the queue contains no elements. The member functions `size()` and `empty()` are provided to check whether the queue contains elements.

If you don't like the standard interface of `queue<>`, you can easily write a more convenient interface. See Section 12.2.3, page 641, for an example.

12.2.2. Example of Using Queues

The following program demonstrates the use of class `queue<>`:

Click here to view code image

```
// contadapt/queue1.cpp

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

int main()
{
    queue<string> q;

    // insert three elements into the queue
    q.push("These ");
    q.push("are ");
    q.push("more than ");

    // read and print two elements from the queue
    cout << q.front();
    q.pop();
    cout << q.front();
    q.pop();

    // insert two new elements
    q.push("four ");
    q.push("words!");

    // skip one element
    q.pop();

    // read and print two elements
    cout << q.front();
    q.pop();
    cout << q.front() << endl;
    q.pop();

    // print number of elements in the queue
    cout << "number of elements in the queue: " << q.size()
         << endl;
}
```

The output of the program is as follows:

```
These are four words!
number of elements in the queue: 0
```

12.2.3. A User-Defined Queue Class

The standard class `queue<>` prefers speed over convenience and safety. This is not what programmers always prefer. But you can easily provide your own queue class as explained according to the user-defined stack class (see Section 12.1.3, page 635). An corresponding example is provided on the Web site of this book in files *contadapt/Queue.hpp* and `contadapt/queue2.cpp`.

12.2.4. Class queue<> in Detail

The queue<> interface maps more or less directly to corresponding container members (see Section 12.1.4, page 637, for the corresponding mapping of class stack<>). See Section 12.4, page 645, for details of the provided members and operations.