Queue implementation in C++

A queue is a linear data structure that serves as a container of objects that are inserted & removed according to the FIFO (first-in first-out) principle.

Queue has three main operations: enqueue, dequeue and peek. We have discussed about these operations in previous post and also covered C implementation of queue data structure using an array and linked list. In this post, we will cover queue implementation in C++ using class and STL.

Below queue implementation in C++ covers below operation –

- `enqueue:` Inserts a new element at the rear of the queue.
- `dequeue:` Removes the front element of the queue.
- `peek:` Returns the front element present in the queue without dequeuing it.
- `isEmpty:` Checks if the queue is empty.
- `isFull:` Checks if the queue is full.
- `size:` Returns the number of elements present in the queue.

Queue Implementation using an array –

```
#include <iostream>
#include <cstdlib>
using namespace std;

// define default capacity of the queue
#define SIZE 10

// Class for queue
class queue
{
    int *arr;       // array to store queue elements
    int capacity;   // maximum capacity of the queue
    int front;      // front points to front element in the qu
```


Browse

Algorithm Amazon Beginner Binary Search Bit Hacks Bottom-up Breadth-first search Depth-first search Easy FIFO Greedy Hard Hashing Intro LIFO Maze Medium Microsoft Must Know Priority Queue Recursive Searching Sliding-Window Top-down Trie

```
public:
    queue(int size = SIZE);     // constructor
    ~queue();                   // destructor

    void dequeue();
    void enqueue(int x);
    int peek();
    int size();
    bool isEmpty();
    bool isFull();
};

// Constructor to initialize queue
queue::queue(int size)
{
    arr = new int[size];
    capacity = size;
    front = 0;
    rear = -1;
    count = 0;
}

// Destructor to free memory allocated to the queue
queue::~queue()
{
    delete[] arr;
}

// Utility function to remove front element from the queue
void queue::dequeue()
{
    // check for queue underflow
    if (isEmpty())
    {
        cout << "UnderFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Removing " << arr[front] << '\n';

    front = (front + 1) % capacity;
    count--;
}

// Utility function to add an item to the queue
void queue::enqueue(int item)
{
    // check for queue overflow
    if (isFull())
    {
        cout << "OverFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }

    cout << "Inserting " << item << '\n';

    rear = (rear + 1) % capacity;
    arr[rear] = item;
    count++;
}

// Utility function to return front element in the queue
int queue::peek()
{
    if (isEmpty())
    {
        cout << "UnderFlow\nProgram Terminated\n";
        exit(EXIT_FAILURE);
    }
    return arr[front];
}

// Utility function to return the size of the queue
```

```
}

// Utility function to check if the queue is empty or not
bool queue::isEmpty()
{
    return (size() == 0);
}

// Utility function to check if the queue is full or not
bool queue::isFull()
{
    return (size() == capacity);
}

int main()
{
    // create a queue of capacity 5
    queue q(5);

    q.enqueue(1);
    q.enqueue(2);
    q.enqueue(3);

    cout << "Front element is: " << q.peek() << endl;
    q.dequeue();

    q.enqueue(4);

    cout << "Queue size is " << q.size() << endl;

    q.dequeue();
    q.dequeue();
    q.dequeue();

    if (q.isEmpty())
        cout << "Queue Is Empty\n";
    else
        cout << "Queue Is Not Empty\n";

    return 0;
}
```

Download Run Code

Output:

```
Inserting 1
Inserting 2
Inserting 3
Front element is: 1
Removing 1
Inserting 4
Queue size is 3
Removing 2
Removing 3
Removing 4
Queue Is Empty
```

The time complexity of all above queue operations is `O(1)`.

Using `std::queue`

and pop_front operations with LIFO semantics. Java's library contains a Queue interface that specifies queue operations.

---

std::queue

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

// Queue implementation in C++ using std::queue
int main()
{
    queue<string> q;

    q.push("A");    // Insert "A" in the queue
    q.push("B");    // Insert "B" in the queue
    q.push("C");    // Insert "C" in the queue
    q.push("D");    // Insert "D" in the queue

    // Returns the number of elements present in the queue
    cout << "Queue size is " << q.size() << endl;

    // Prints the front of the queue ("A")
    cout << "Front element is: " << q.front() << endl;

    // Prints the rear of the queue ("D")
    cout << "Rear element is: " << q.back() << endl;

    q.pop();    // removing the front element ("A")
    q.pop();    // removing the next front element ("B")

    cout << "Queue size is " << q.size() << endl;

    // check if queue is empty
    if (q.empty())
        cout << "Queue is Empty\n";
    else
        cout << "Queue is not Empty\n";

    return 0;
}
```

Download Run Code

std::list

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

// Queue implementation in C++ using std::list
int main()
{
    list<string> q;

    q.push_back("A");    // Insert "A" in the queue
    q.push_back("B");    // Insert "B" in the queue
    q.push_back("C");    // Insert "C" in the queue
    q.push_back("D");    // Insert "D" in the queue

    // Returns the number of elements present in the queue
    cout << "Queue size is " << q.size() << endl;

    // Prints the front of the queue ("A")
    cout << "Front element is: " << q.front() << endl;

    // Prints the rear of the queue ("D")
```

```
25      q.pop_front();    // removing the next front element ("B")
26
27      cout << "Queue size is " << q.size() << endl;
28
29      // check if queue is empty
30      if (q.empty())
31          cout << "Queue is Empty\n";
32      else
33          cout << "Queue is not Empty\n";
34
35      return 0;
36  }
```

Download Run Code

Output:

```
Queue size is 4
Front element is: A
Rear element is: D
Queue size is 2
Queue is not Empty
```

Also See:

Circular Queue implementation in C

A queue is a linear data structure that serves as a collection of elements, with three main operations: Enqueue operation, which adds an element to the rear position in the queue. Dequeue operation, which removes an element from the front position in the queue. Peek or front operation, which returns the front element without dequeuing ... Continue reading

td Techie Delight 0

Queue Implementation in Java

A queue is a linear data structure which follows the FIFO (first-in first-out) principle. That means the object which is inserted first will be the first one out, followed by the object which was inserted next. The queue supports the following core operations – enqueue: Inserts an item at the rear of the queue. ... Continue reading

td Techie Delight 0

Queue Implementation in Python

A queue is a linear data structure that follows the FIFO (First-In First-Out) order i.e. the item which is inserted first will be the first one out. A Queue supports the following standard operations: append: Inserts an element at the rear (right side) of the queue. pop: Removes the element from the front (left ... Continue reading

Techie Delight 0

Queue Implementation using Linked List – C, Java and Python

A queue is an linear data structure that serves as a collection of elements, with three main operations: enqueue, dequeue and peek. We have discussed about these operations in previous post and covered array implementation of queue data structure. In this post, linked list implementation of queue is discussed. A queue can be easily implemented using ... Continue reading

Techie Delight 0

(**20** votes, average: **4.40** out of 5)

Thanks for reading.

Sharing is caring:

Tweet Share 0 Share

Please use our online compiler to post code in comments using C, C++, Java, Python, JavaScript, C#, PHP and many more popular programming languages.

Like us? Refer us to your friends and help us grow. Stay Safe. Happy coding :)

← Generate power set of a given set

Queue Implementation using Templates in C++ →

Leave a Reply

b i b-quote u ul ol li code

Join the discussion (Please use an online compiler to post code in comments)

▲ newest ▲ oldest ▲ most voted

Michael

Guest

Cool! Can you explain why you use the modulo operator when computing the new front?

```
1 | rear = (rear + 1) % capacity;
```

Thanks

2 Reply 1 year ago

Aditya

Guest

To make queue circular.

3 Reply 1 year ago

Alan Gao

Guest

Interesting...you don't pre-allocate storage space but directly use arr[rear] to insert items. And it works?
int *arr;
arr[rear] = item;
Here who is responsible for the storage, or memory allocation?

1 Reply 11 months ago

Admin

Author

Thanks for sharing your concerns. Please have another look at the code. The storage is allocated inside the constructor for the queue class.

0 Reply 11 months ago

Alan Gao

Guest

I see, you pre-allocate the max space in the constructor, but it defeats the purpose of a queue – the capacity of a queue grows and shrinks from time to time and you

prefer deque due to its dynamic space allocation and better performance for push and pop. For those that prefer accessing items via index, vector is a better choice because it mostly assert a continuous memory address allocation.

1 Reply 11 months ago

Guest

Rithik Singh

Why not use template ... so that we can get a more generalized form of queue...
Please repost this with these considerations...

0 Reply 1 year ago

Guest

Yuvi

delete[] should be used in the destructor of Queue class (on line 42) instead of simple delete operator. Otherwise, it's memory leak.

0 Reply 3 months ago

Author

Admin

Thanks for bringing this to our notice. We have updated the code. Happy coding ☺

0 Reply 3 months ago

Guest

Ataj

I did not understand this. How does this help in the dequeue process? Can someone please explain.

front = (front + 1) % capacity;

0 Reply 2 months ago