

# CP::queue

Will the circle be unbroken?

# Intro

- Queue, unlike stack, require more sophisticated technique to achieve fast performance
- We start by writing a simple class that just work (slowly)
- Then we try to improve it

# Key Idea

- Just like stack, we will use the same format as vector, using dynamic array to store data
- However, we have to somehow manage how we works with front() and back() of the queue

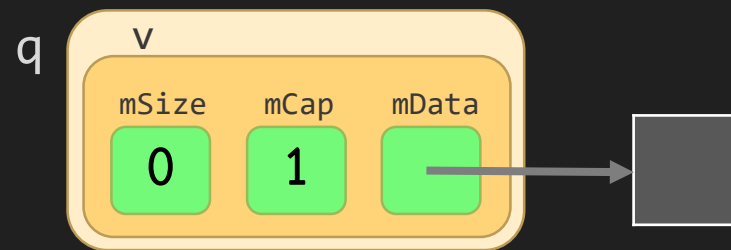
# v0.1 simple implementation of queue

- To illustrate this idea, we will use a `vector` as our data member
- `push(e)` is simply `v.push_back(e)`, this is fast
- `front()` is `v[0]`, `back()` is `v[v.size()-1]`, this is also fast
- `pop()` is `v.erase(v.begin())`, this is slow (always proportional to `v.size()`)
  - Unlike `std::queue` which has very fast `pop()`

```
namespace CP {
    template <typename T>
    class queue {
    protected:
        std::vector<T> v;
    public:
        //----- capacity function -----
        queue() : v() {}
        //----- capacity function -----
        bool empty() const { return v.empty();}
        size_t size() const { return v.size();}
        //----- access -----
        const T& front() const { return v[0];}
        const T& back() const { return v[v.size()-1];}
        //----- modifier -----
        void push(const T& element) { v.push_back(element);}
        void pop() { v.erase(v.begin());}

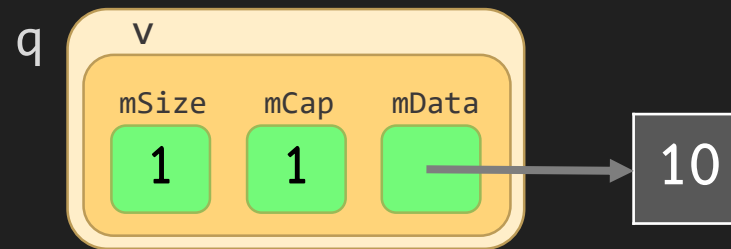
    };
}
```

```
CP::queue<int> q;
```

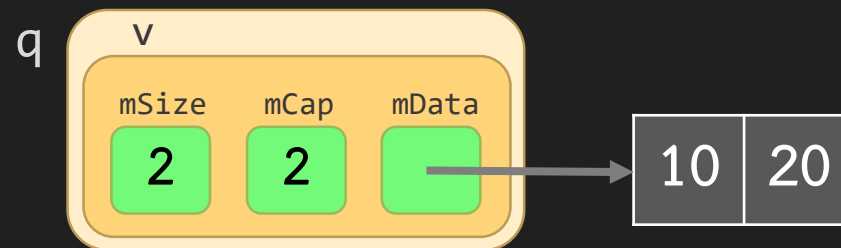


v0.1 example

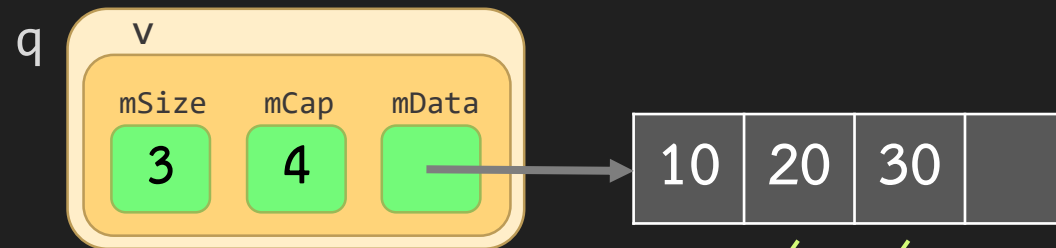
```
q.push(10);
```



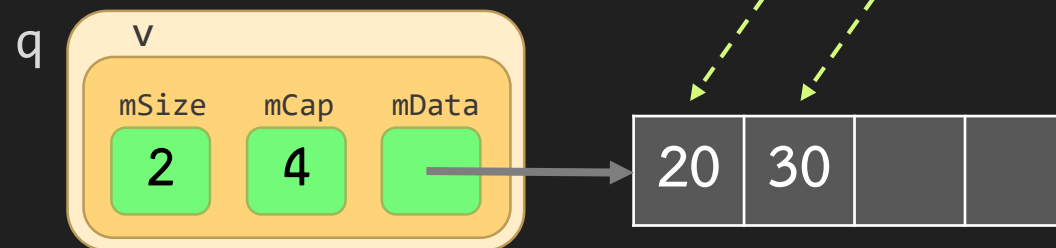
```
q.push(20);
```



```
q.push(30);
```



```
q.pop();
```



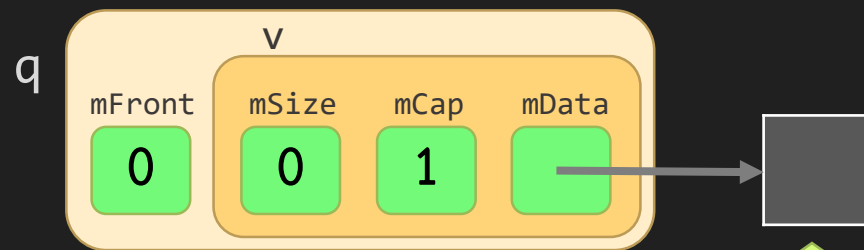
## v0.2 faster queue

- Add more data member `mFront`, initialized as 0
- `push(e)` is simply `v.push_back(e)`, this is fast
- `front()` is `v[mFront]`, `back()` is `v[v.size()-1]`, this is also fast
- `pop()` is `mFront++`, this is fast
  - However, we don't really remove anything when pop

```
#include <vector>

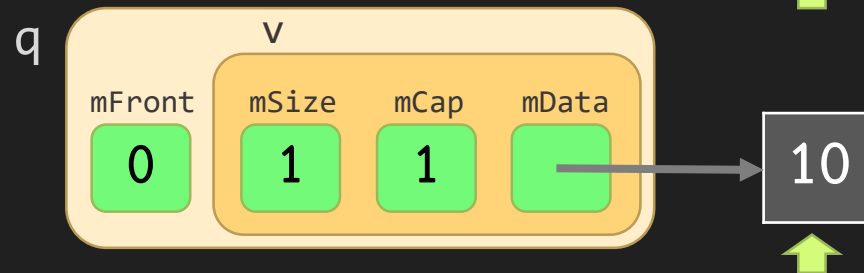
namespace CP {
    template <typename T>
    class queue
    {
    protected:
        std::vector<T> v;
        int mFront;
    public:
        //----- capacity function -----
        queue() : v(), mFront() {}
        //----- capacity function -----
        bool empty() const { return v.empty();}
        size_t size() const { return v.size()- mFront;}
        //----- access -----
        const T& front() const { return v[mFront];}
        const T& back() const { return v[v.size()-1];}
        //----- modifier -----
        void push(const T& element) { v.push_back(element);}
        void pop() { mFront++;}
    };
}
```

```
CP::queue<int> q;
```



v0.2 example

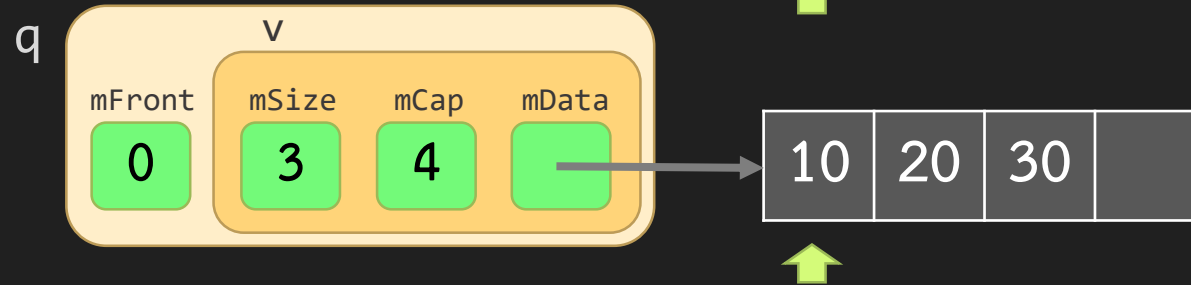
```
q.push(10);
```



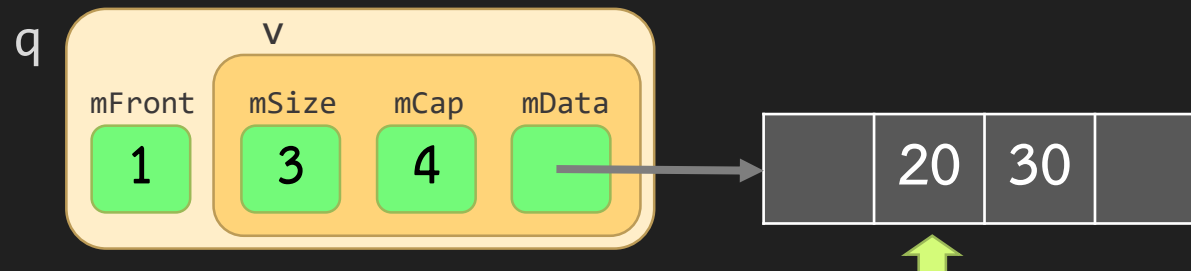
```
q.push(20);
```



```
q.push(30);
```



```
q.pop();
```

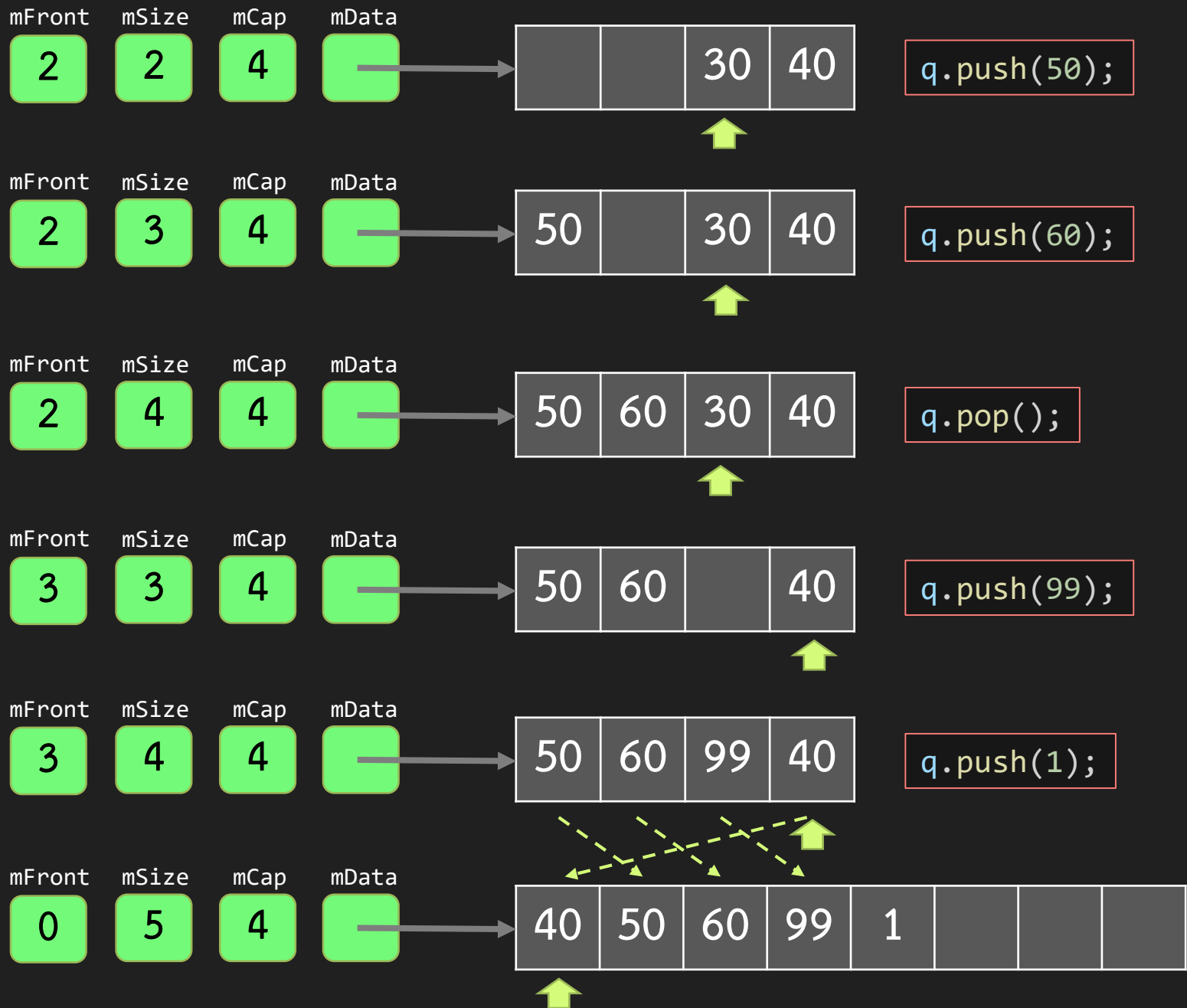


# Problem with v0.2

- Fast but use too many space
- Queue grows according to how many time push is called
  - regardless of how many pop is called
- The data stored in the vector can be much larger than the actual data in the queue
- Does not really work in real world

```
for (int i = 0; i < 1000000; i++) {  
    q.push(i);  
    q.pop();  
}  
std::cout << q.size() << std::endl;
```



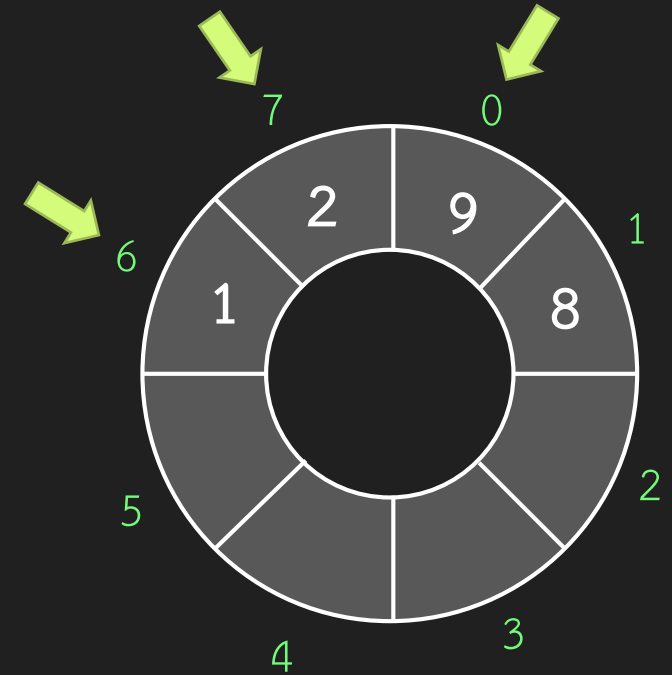
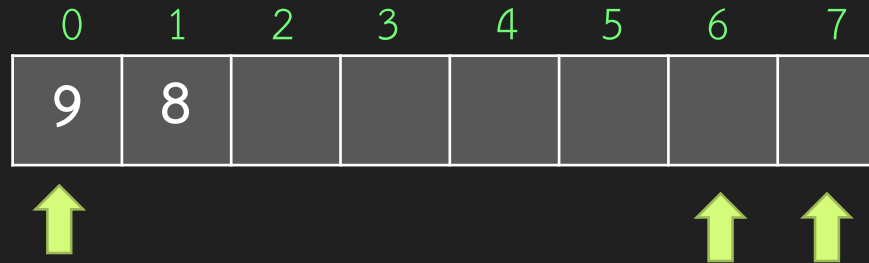


## Final Idea

- We take v0.2 and **reuse** the area at the beginning of `mData`
  - Expand when necessary
  - Re-arrange when expand

# Circular Queue

- We can think of `mData` to be circular
  - End of the last element of the `mData` is connected to the first element
- Consider  $i^{\text{th}}$  element
  - the next element is  $(i+1) \% \text{mCap}$
  - The previous element is  $(i-1+\text{mCap}) \% \text{mCap}$
  - Next k element is  $(i+k) \% \text{mCap}$



# queue.h

Almost the same  
but have to take  
care of mFront

Same as vector

Circular queue  
implementation

```
namespace CP {
    template <typename T>
    class queue
    {
        protected:
            T *mData;
            size_t mCap;
            size_t mSize;
            size_t mFront;
            void expand(size_t capacity) {...}
            void ensureCapacity(size_t capacity) {...}
        public:
            //----- constructor -----
            queue(const queue<T>& a) {...}
            queue() {...}
            queue<T>& operator=(queue<T> other) {...}
            ~queue() {...}
            //----- capacity function -----
            bool empty() const {...}
            size_t size() const {...}
            //----- access -----
            const T& front() const {...}
            const T& back() const {...}
            //----- modifier -----
            void push(const T& element) {...}
            void pop() {...}
    };
}
```

Additional data  
member **mFront**

```

template <typename T>
class queue {
protected:
    T *mData;  size_t mCap;  size_t mSize;  size_t mFront;
public:
    // default constructor
    queue() : mData(new T[1]()), mCap(1),
             mData[0], mFront(0) { }
    // copy constructor
    queue(const queue<T>& a) : mData(new T[a.mCap]()), mCap( a.mCap ),
                             mData[0], mFront( a.mFront ) {
        for (size_t i = 0; i < a.mCap; i++) {
            mData[i] = a.mData[i];
        }
    }
    // copy assignment operator
    queue<T>& operator=(queue<T> other) {
        using std::swap;
        swap(mSize, other.mSize);
        swap(mCap, other.mCap);
        swap(mData, other.mData);
        swap(mFront, other.mFront);
        return *this;
    }
    ~queue() {
        delete [] mData;
    }
};

```

List initialization

Need to copy entire mData  
(not just mSize)

Also swap mFront

same

## Ctor, Dtor, copy

- Dtor is the same
- ctor also have to initialize mFront
- Copy also have to copy mFront

# front(), back(), pop()

```
template <typename T>
class queue {
protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    size_t mFront;
public:
    //----- access -----
    const T& front() const {
        return mData[mFront];
    }
    const T& back() const {
        return mData[(mFront + mSize - 1) % mCap];
    }
    //----- modify -----
    void pop() {
        mFront = (mFront + 1) % mCap;
        mSize--;
    }
};
```

- $\text{back} = \text{mFront} + \text{mSize} - 1$ 
  - Also circular (by  $\% \text{mCap}$ )
- $\text{pop} = \text{move mFront by 1}$ 
  - Also circular
  - Also change size

# push, expand

- push add data to  $(mFront + mSize) \% mCap$ 
  - The space just after `back()`
- Expand re-pack the `mData` so that `mFront` is 0
- `ensureCapacity` is the same

```
template <typename T>
class queue {
protected:
    T *mData;
    size_t mCap;
    size_t mSize;
    size_t mFront;
    void expand(size_t capacity) {
        T *arr = new T[capacity]();
        for (size_t i = 0; i < mSize; i++) {
            arr[i] = mData[(mFront + i) % mCap];
        }
        delete [] mData;
        mData = arr;
        mCap = capacity;
        mFront = 0;
    }
    void ensureCapacity(size_t capacity) {
        if (capacity > mCap) {
            size_t s = (capacity > 2 * mCap) ? capacity : 2 * mCap;
            expand(s);
        }
    }
public:
    void push(const T& element) {
        ensureCapacity(mSize+1);
        mData[(mFront + mSize) % mCap] = element;
        mSize++;
    }
};
```

# Analysis

- All access, modification is fast (constant time)
- Space is re-used
  - It is not shrunk when mSize reduce
  - Space is not more than double of maximum mSize during its lifetime

# Exercise

- We implement circular queue by maintain `mFront` and use circular logic (`% mCap`) to calculate the position of back of the queue
  - Can we maintain `mBack` instead?
  - Can we maintain both `mFront` and `mBack` but not `mSize`?
- How about `mCap`, if we know `mFront`, `mSize`, `mBack`, can we calculate `mCap`?

mFront	mSize	mBack	front()	back()	size()
YES	YES	No	v[mFront]	v[(mFront + mSize - 1) % mCap]	mSize
No	YES	YES	????	v[mBack]	mSize
YES	No	YES	v[mFront]	v[mBack]	????



# Now, meet deque

- Can you modify queue to include
  - `push_front()`, add to the front of the queue
  - `pop_back()`, remove from back of the queue
- All operation should still be constant time