# Stack and Queue

A **stack** is a finite sequence of (zero or more) elements $(x_1 , ... , x_n )$

Two operations, **push** and **pop** are defined on a stack.
**push($x_{n+1}$)** changes a stack $(x_1 , ... , x_n )$ to $(x_1 , ... , x_n , x_{n+1})$

**pop()** changes a nonempty stack $(x_1 , ... , x_n , x_{n+1})$ to $(x_1 , ... , x_n )$
and also returns the element $x_{n+1}$

**Example** Start with an empty stack of integers, and perform the following operations.

```
push(5)
push(9)
push(2)
pop()
pop()
push(4)
pop()
pop()
```

What values are returned by pop() operations ?

A **stack** is a finite sequence of (zero or more) elements $(x_1, \ldots, x_n)$

Two operations, **push** and **pop** are defined on a stack.
**push($x_{n+1}$)** changes a stack $(x_1, \ldots, x_n)$ to $(x_1, \ldots, x_n, x_{n+1})$

**pop()** changes a nonempty stack $(x_1, \ldots, x_n, x_{n+1})$ to $(x_1, \ldots, x_n)$
and also returns the element $x_{n+1}$

**Example** Start with an empty stack of integers, and perform the following operations.

```
push(5) () → (5)
push(9) (5) → (5,9)
push(2) (5,9) → (5,9,2)
pop()   (5,9,2) → (5,9) and 2 is returned
pop()   (5,9) → (5)     and 9 is returned
push(4) (5) → (5,4)
pop()   (5,4) → (5)     and 4 is returned
pop()   (5) → ()        and 5 is returned
```

What values are returned by pop() operations ? 2 9 4 5

A **queue** is a finite sequence of (zero or more) elements ($x_1$ , ... , $x_n$ )

Two operations, **enqueue** and **dequeue** are defined on a stack.
**enqueue($x_{n+1}$)** changes a stack ($x_1$ , ... , $x_n$ ) to ($x_1$ , ... , $x_n$ , $x_{n+1}$)

**dequeue()** changes a nonempty stack ($x_1$ , $x_2$ , ... , $x_n$ ) to ($x_2$ , ... , $x_n$ )
and also returns the element $x_1$

**Example** Start with an empty stack of integers, and perform the following operations.

```
enqueue(5) () → (5)
enqueue(9) (5) → (5,9)
enqueue(2) (5,9) → (5,9,2)
dequeue()  (5,9,2) → (9,2) and 5 is returned
dequeue()  (9,2) → (2)     and 9 is returned
enqueue(4) (2) →(2,4)
dequeue()  (2,4) →(4)      and 2 is returned
dequeue()  (4) → ()        and 4 is returned
```

What values are returned by dequeue() operations ? 5 9 4 2

# Implementing Stack and Queue Using Array

## Implementing a stack (of integers) using array

Use an integer array named **s** of size MAX to store the stack elements.

s[0], s[1] , ... stores the elements $x_1$ , $x_2$ , ...

Notice that: (i) We cannot store more than MAX elements.
(ii) When we store only a few elements space is wasted.

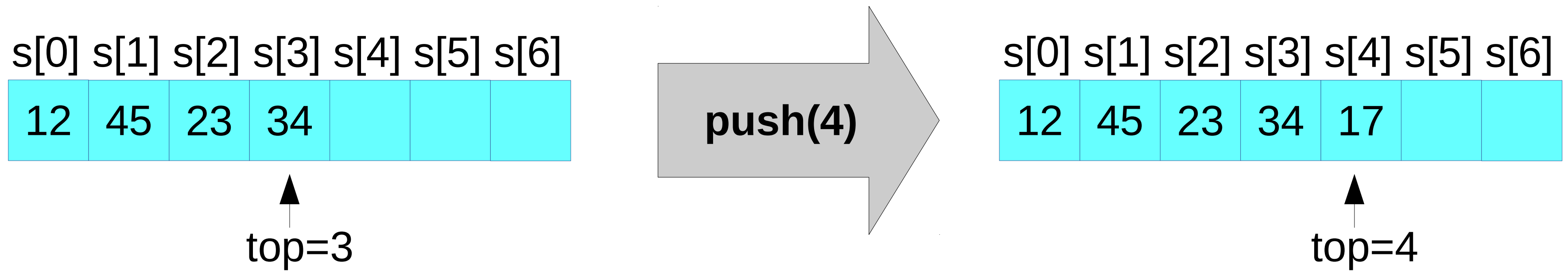
Use an integer named **top** to store the index of the last element.

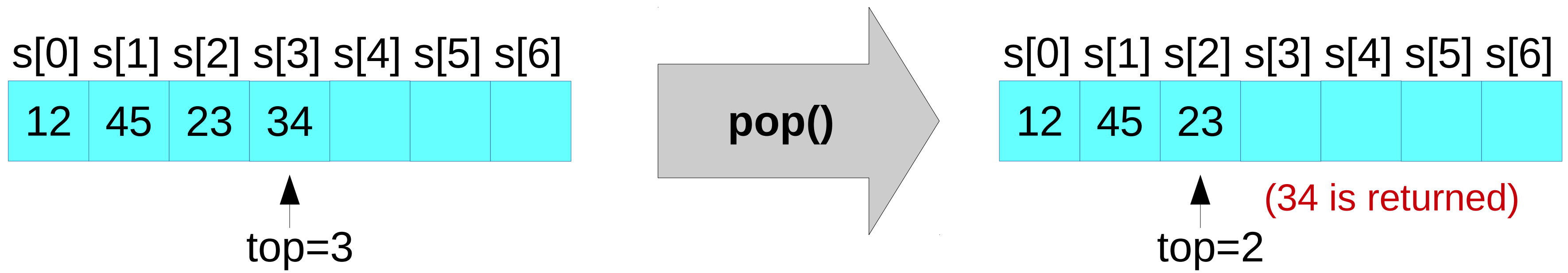
Notice that: If the stack contains **n** elements then **`top = n-1`**
To initialize the stack as empty we set **`top = -1`**

During **`push(x)`** increment **`top`** by 1 and store **`x`** in **`s[top]`**

During **`pop()`** let t = s[top], decrement **`top`** by 1 and return t.

**Example: where MAX = 7**



```
class stack
{
   public:
   void push(int);
   int pop(void);
   bool isempty(void);
   bool isfull(void);
   void show(void);

   stack(int);
   ~stack(void);

   private:
   int MAX, *s, top;
};
```

returns true if and only if
the stack is empty i.e. **top is -1**

returns true if and only if
the stack is full i.e. **top is MAX-1**

**s** should point to the beginning of the array

**<u>s</u>** <u>is just a pointer</u>, we must allocate space
for array elements using dynamic allocation

When an object of this class is created we should,
(i) initialise MAX
(ii) allocate space for array elements s[0],..,s[MAX]
(iii) initialise top to -1

When an object of this lass is destroyed we should,
de allocate space for array elements s[0],..,s[MAX]

```
class stack
{
    public:
    void push(int);
    int pop(void);
    bool isempty(void);
    bool isfull(void);
    void show(void);

    stack(int);
    ~stack(void);

    private:
    int MAX, *s, top;
};
```

**The constructor takes an argument !!**
(constructors are automatically called when an object is created )
**How do we pass the argument ??**

```
stack::stack(int size)
{
  MAX = size;
  s = new int[MAX];
  top = -1;
}
```

When an object of this class is created we should,
(i) initialise MAX
(ii) allocate space for array elements s[0],..,s[MAX]
(iii) initialise top to -1

When an object of this lass is destroyed we should,
de allocate space for array elements s[0],..,s[MAX]

```
stack::~stack(void)
{
  delete[] s;
}
```

```
class stack
{
    public:
    void push(int);
    int pop(void);
    bool isempty(void);
    bool isfull(void);
    void show(void);

    stack(int);
    ~stack(void);

    private:
    int MAX, *s, top;
};
```

**The constructor takes an argument !!**
(constructors are automatically called when an object is created )
**How do we pass the argument ??**

```
stack::stack(int size)
{
  MAX = size;
  s = new int[MAX];
  top = -1;
}
```

When we create an object of this class we must use the following syntax.
**`stack`** <object_name>**(**<an integer>**)**

**For example**

```
int main()
{
  stack st(3);
  st.push(6); st.push(3); st.push(7);
  if(st.isfull()) cout << "stack full";
}
```

This creates an object of stack class with array size 3

This returns **true**

In general if we have a class where the constructor takes n arguments,

```
class <class_name>
{
   ...
   <class_name>(<type_1>, ... ,<type_n>);
   ...
};
```

then to create object of this class we must use the following syntax

```
<class_name>    <object_name>(<arg_1>, ... ,<arg_n>)
```

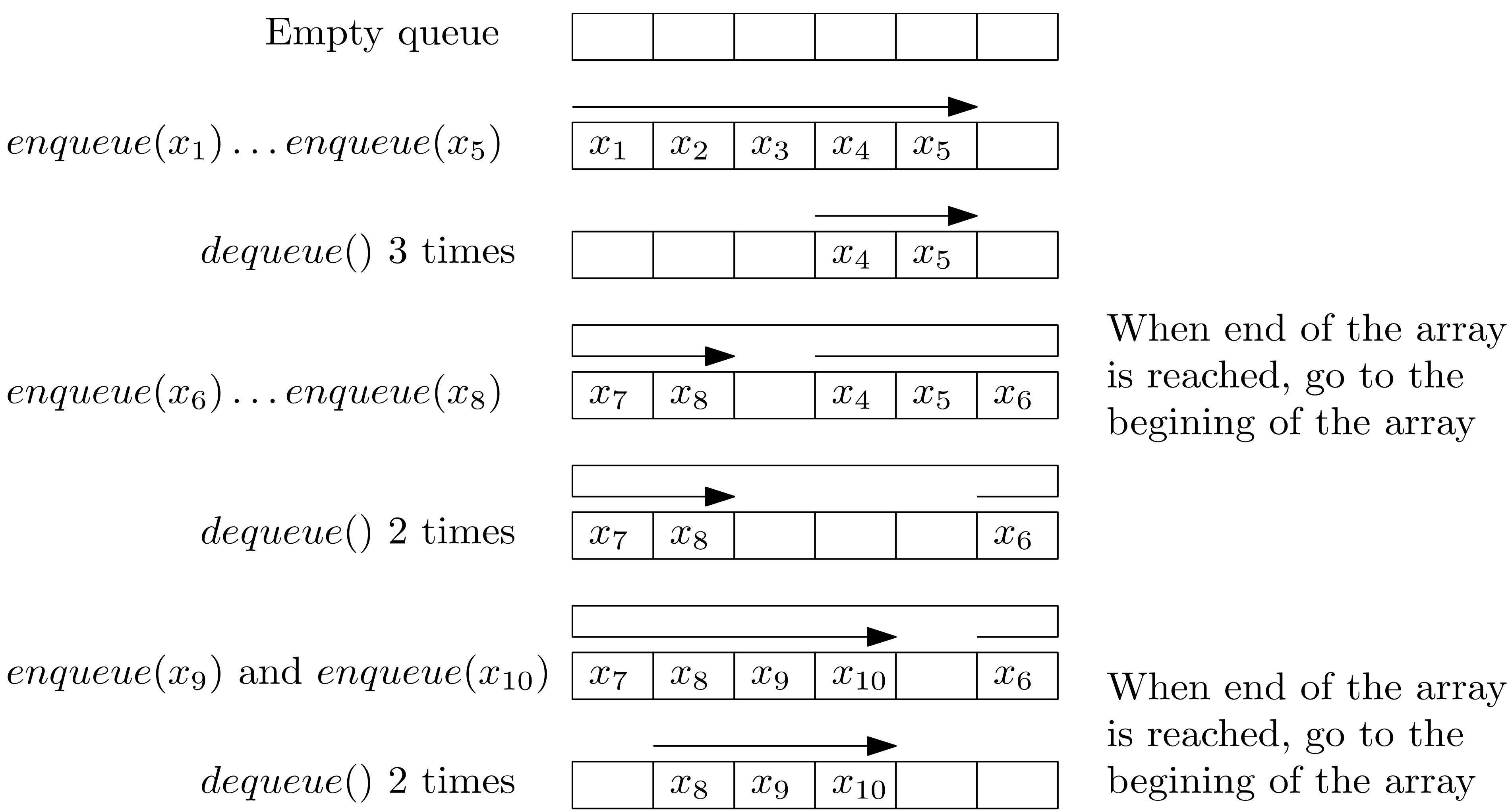
where <arg_1> is of <type_1> , <arg_2> is of <type_2> ... and so on.

## Implementing Queue (of integers) using array

Use an integer array, say q[MAX] , to store the queue elements, where MAX is a large integer.

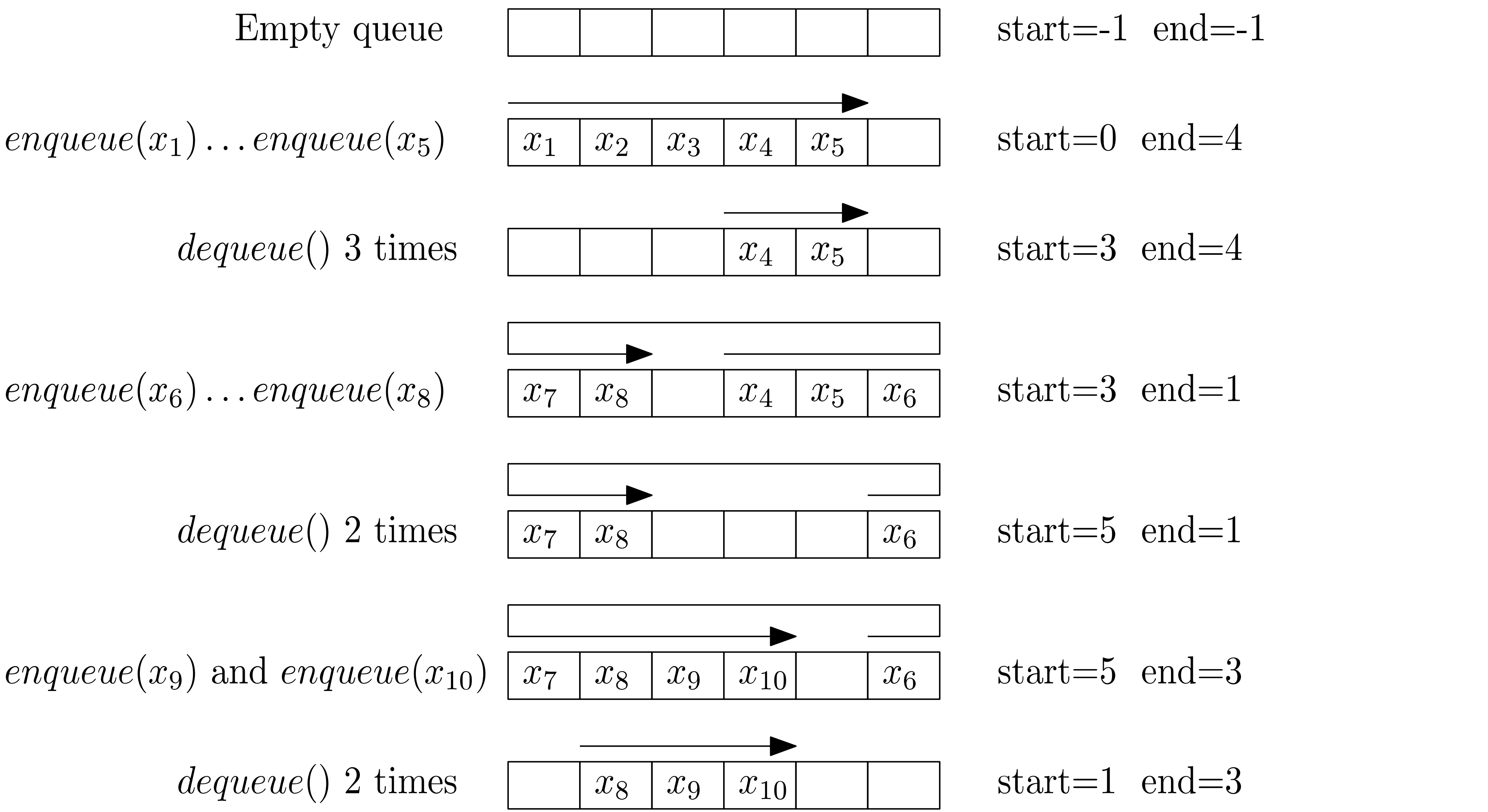
Notice that: (i) We cannot store more than MAX elements.
(ii) When we store only a few elements space is wasted.

Here is an example illustrating the idea where MAX=6.



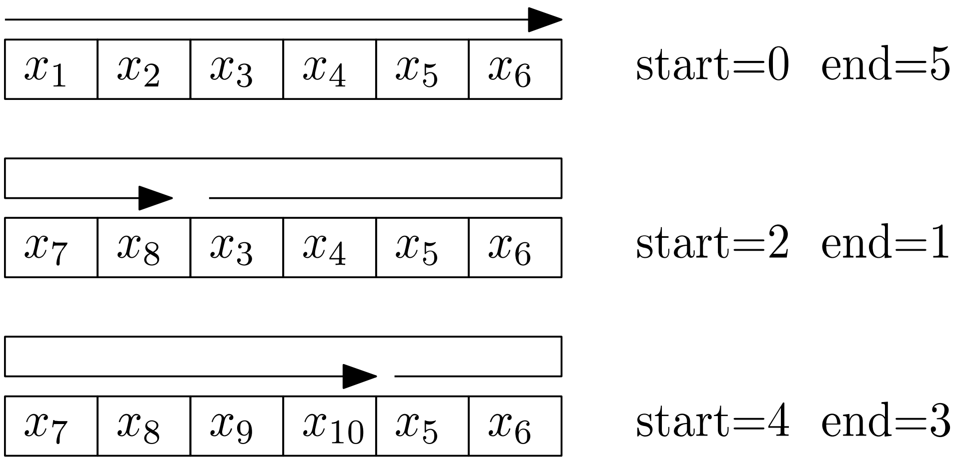
Use an integer array, say **q[MAX]** , to store the queue elements, where **MAX** is a large integer. Use two integer variable **`start`** and **`end`** to store the starting and ending position of queue.

- For <u>empty queue</u> we set **`start=-1`** and **`end=-1`**
- <u>enqueue</u> changes **`end`** to **`(end+1)%MAX`** (if the list is initially empty set start=end=0)
- <u>dequeue</u> changes **`start`** to **`(start+1)%MAX`**

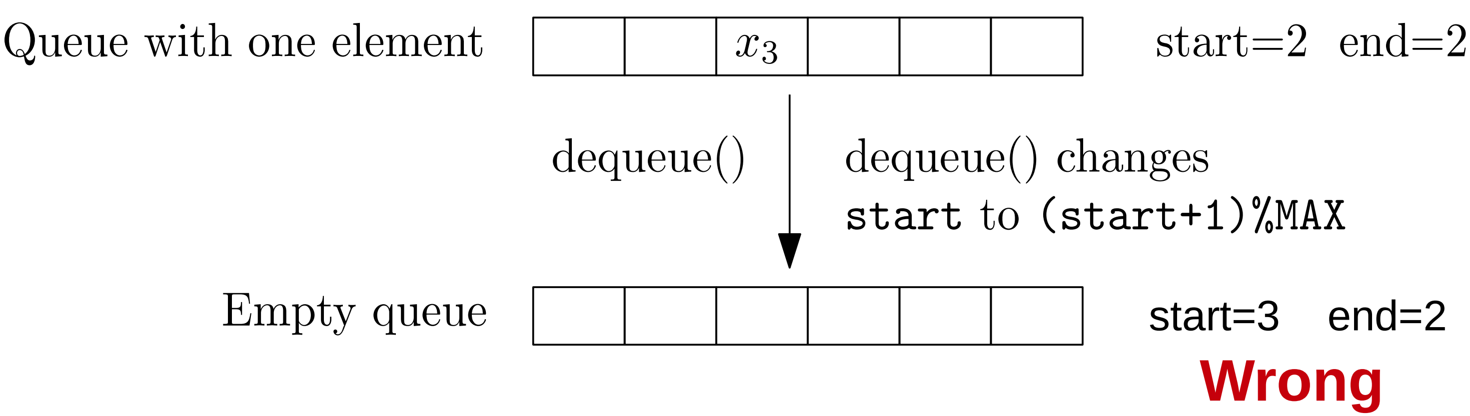


The queue is **full** if and only if `(end+1)%MAX` is `start`.

Example




If **dequeue makes the queue empty**, we explicitly set `start=-1` and `end=-1`. Otherwise the following may happen.



```
class queue
{
    public:
    void enqueue(int);
    int dequeue(void);
    bool isempty(void);
    bool isfull(void);
    void show(void);

    queue(int);
    ~queue(void);

    private:
    int MAX,*q,start,end;
};
```

enqueue(x) changes **end to (end+1)%MAX**
and stores **x at q[end]**
( if the queue is initially empty explicitly set start=0 and end=0 )

dequeue(x) **returns q[start]**
also changes **start to (start+1)%MAX**
( if the queue becomes empty explicitly set start=-1 and end=-1 )

returns true if and only if the queue is empty
i.e. **start is -1 and end is -1**

returns true if and only if the queue is full
i.e. **(end+1)%MAX is start**

**q** should point to the beginning of the array

**q** is just a pointer, we must allocate space
for array elements using dynamic allocation

When an object of this class is created we should,
(i) initialise MAX
(ii) allocate space for array elements q[0],..,q[MAX]
(iii) initialise start and end to -1

When an object of this lass is destroyed we should,
de allocate space for array elements q[0],..,q[MAX]

## Inheritance

In Object Oriented Programming,
we know that the user of a class can use the functionalities of the class
without knowing / bothering about the implementation.

OOP also provides a mechanism so that,
user can create his own class which **inherits** functionalities of a pre-existing class
without knowing / bothering about the implementation of the original class.

This is known as **inheritance.**
The pre-existing class is called **base-class**.
The newly created class is called the **derived class**.

Suppose we already have a class B with 20 public member functions.

class B is defined inside header B.h

**B.h**

```
class B
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);
  private:
  ...
};
```

B.h gives only the declarations of member functions

We do not know anything about the implementation of the member functions `func1() ... func20()`

We can still use the class by including the header and creating object.

**TestB.cpp**

```
#include "B.h"
int main()
{
  B t;
  t.func1(5);
  t.func2();
  ...
  t.func20();
};
```

Suppose we already have a class B with 20 public member functions.

class B is defined inside header B.h

B.h gives only the declarations of member functions

We do not know anything about the implementation of the member functions `func1() ... func20()`

**B.h**

```
class B
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);
  private:
  ...
};
```

We can still use the class by including the header and creating object.

**TestB.cpp**

```
#include "B.h"
int main()
{
  B t;
  t.func1(5);
  t.func2();
  ...
  t.func20();
};
```

---

**Now we want to create a new class D s.t.**

- D also has the member functions func1()...func20()
- Additionally, D has another member function foo() (we shall define foo() ourselves)
- We shall use class D by creating objects and calling the member functions

```
<definition of class D>
<definition of D::foo() >
int main()
{
  D x;
  x.func1(5); x.func2();
  ...
  x.foo();
};
```

## A crude solution (DO NOT do this)

```
class D
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);

  void foo(void)
};
```

Copy-Paste

```
class B
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);
  private:
  ...
};
```

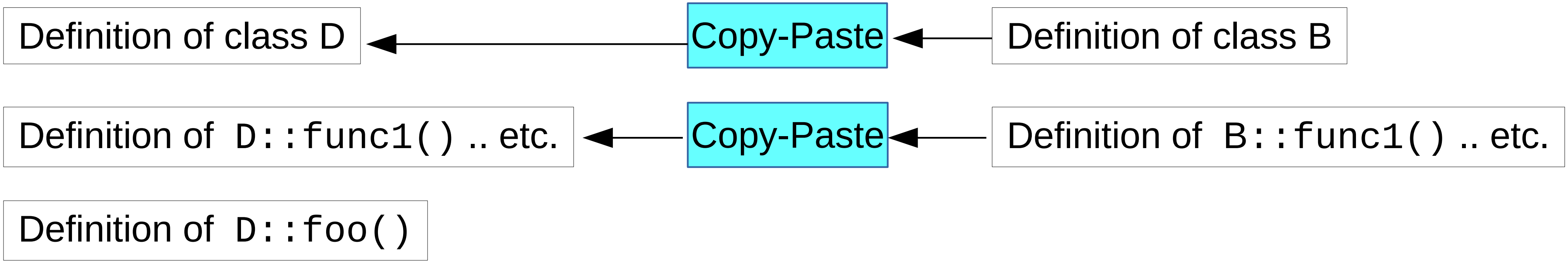
```
void D::func1(int i)
{
  <implementation of func1>
}
..............
..............
void D::func20(void)
{
  <implementation of func20>
}

void D::foo(void)
{ cout<<"hello"; }
```

Copy-Paste

........
........

Copy-Paste

```
void B::func1(int i)
{
  <implementation of func1>
}
..............
..............
void B::func20(void)
{
  <implementation of func20>
}
```




**Problem 1:**

Source code for B::func1() etc. may not be available.
It could be proprietary and may come as a precompiled binary file.
Developer of class B may not allow you to know the actual implementation.

Even when you have access to the code,

**Problem 2:**

Either you need to understand how those functions work,
or you have a chunk of code in your own program and you have no idea what it is doing.

**A better solution:** Derive a class D from the base class B and inherit its members.

class B is defined inside header B.h

**B.h**

```
class B
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);
  private:
  ...
};
```

We may define our class D as follows

**D.h**

```
#include "B.h"
class D : public B
{
  public:
  void foo(void);
};
```

We need to define our own member foo()

**D.cpp**

```
void B::foo(void)
{ cout<<"hello"; }
```

Finally we can use class D as intended

```
#include "D.h"
int main()
{
  D x;
  x.func1(5);
  x.func2();
  ...
  x.foo();
};
```

We do not need to know anything about the implementation of the base class

class B is defined inside header B.h

**B.h**

```
class B
{
  public:
  void func1(int);
  int func2(void);
  ...
  ...
  void func20(void);
  private:
  ...
};
```

We may define our class D as follows

**D.h**

```
#include "B.h"
class D : public B
{
  public:
  void foo(void);
};
```

We need to define our own member foo()

**D.cpp**

```
void B::foo(void)
{ cout<<"hello"; }
```

Finally we can use class D as intended

```
#include "D.h"
int main()
{
  D x;
  x.func1(5);
  x.func2();
  ...
  x.foo();
};
```

Syntax for public inheritance

Public members (both data and function) of B automatically becomes public members of D

**Private members of B are NOT inherited**

We do not need to know anything about the implementation of the base class

In C++ we have 3 kinds of inheritance - public, private and protected. For all these kinds, **Private members of base class are never inherited**.

We may add new data members or member functions to the derived class

Public Inheritance

To derive a class <der> from a base class <base> we may use the syntax

```
class <der> : public <base>
{
   ....
   ....
};
```

**All the public members of base class become public members of derived class.**

Private Inheritance

To derive a class <der> from a base class <base> we may use the syntax

```
class <der> : private <base>
{
   ....
   ....
};
```

**All the public members of base class become private members of derived class.**

# Implementing Stack and Queue Using Linked List

**Implementing a stack (of integers) using linked list**

Use a linked list to store the stack elements

push(x) should be same as push_back(x)

pop() should be same as pop_back()

**How can we make a new class which re-uses some functionalities of a pre-existing class ?**

**Implementing a stack (of integers) using linked list**

Use a linked list to store the stack elements

push(x) should be same as push_back(x)

pop() should be same as pop_back()

Copy Paste ??

```
class stack
{
   ...
};
```

Copy & Paste

```
class list
{
   ...
};
```

```
void stack::push(int x)
{
   ...
}
```

Copy & Paste

```
void list::push_back(int x)
{
   ...
}
```

```
int stack::pop(void)
{
   ...
}
```

Copy & Paste

```
int list::pop_back(void)
{
   ...
}
```

## Implementing a stack (of integers) using linked list

Use a linked list to store the stack elements

push(x) should be same as push_back(x)

pop() should be same as pop_back()

We may derive our class from previously defined class for linked list

**list.h**

Contains definition of `class` **`list`**
Does not contain implementation

**stack.h** (definition of class stack)

```
#include "list.h"

class stack : public list
{
    public:
    void push(int);
    int pop(void);
};
```

Implementation of `class` **`list`**

May be available as source code, for example **`list.cpp`**

may be available as precompiled binary, for example **`list.o`**

**stack.cpp** (implementation of class stack)

```
#include "stack.h"

void stack:: push(int x)
{ push_back(x); }

int stack::pop(void)
{ return pop_back(); }
```

# Compilation

**`list.h`** Contains definition of class list

**`list.cpp`** or **`list.o`** Contains implementation of class list
(or some other form of binary)

**`stack.h`** Contains definition of class stack

**`stack.cpp`** Contains implementation of class stack

**`main.cpp`** Contains main() function which uses class stack

**Compilation:** `g++ main.cpp stack.cpp list.cpp` (if list.cpp is available)

or `g++ main.cpp stack.cpp list.o` (if list.o is available)

**Implementing a queue (of integers) using linked list**

Use a linked list to store the queue elements

enquequ(x) should be same as push_back(x)

dequeue() should be same as pop_front()

You may derive a class for queue using class list as base class