

# Lecture 21: Template & Container

---

Template: Code reuse; polymorphism

Container of pointers: 1 invariant & 3 rules

- *At-most-once invariant*: any object can be linked to at most one container at any time through pointer.
1. *Existence*: An object must be dynamically allocated before a pointer to it is inserted.
  2. *Ownership*: Once a pointer to an object is inserted, that object becomes the property of the container. No one else may use or modify it in any way.
  3. *Conservation*: When a pointer is removed from a container, either the pointer must be inserted into some container, or its referent must be deleted.

```
template <class T>
class Set {
    T **elts; // An array of T *
    int numElts;
    int sizeElts;
public:
    bool isEmpty();
    void insert(T *vp);
    T *remove();
    Set(int sizeElts = MAXELTS);
    Set(const Set &s);
    Set &operator=(const Set &s);
    ~Set();
};

// where needs to put <T>
Set<T>::Set(){...}
Set<T>::Set(const Set<T> &s){...}
Set<T> &Set<T>::operator=(const Set<T> &s){...}
Set<T>::~~Set(){...}
```

Which invariant/rule is violated?

```
// 1
Set<T>::Set(const Set<T> &s){
    for(int i = 0; i < s.numElts; ++i){
        insert(s[i]);
    }
}

// 2
Set<T>::~~Set(){
```

```

        while(numElts > 0){
            remove();
        }
    }

    // -----
    Set<int> s1;
    Set<int> s2;

    int *v1;
    int *v2 = new int(5);
    int *v3 = new int(6);

    // 3
    s1.insert(v1);
    for(int i = 0; i < 3; ++i){
        s2.insert(&i);
    }

    // 4
    s1.insert(v2);
    delete v2;

    // 5
    int *v3 = new int(*(s1.remove()));
    s2.insert(v3);

```

Answer:

1. At-most-once invariant
2. Conservation rule
3. Existence rule
4. Ownership rule
5. Conservation rule

## Polymorphic Container

```

class BigThing : public Object {
    ...
};

Set<Object> s;
BigThing *bp1 = new BigThing;
s.insert(bp1); // Legal due to substitution rule

Object *op;
BigThing *bp2;
op = s.remove();
bp2 = dynamic_cast<BigThing *>(op);
// if the actual type of `op` is either pointer to `BigThing` or some
// pointer to derived class of `BigThing`, `bp2` will be assigned a pointer

```

```
to `BigThing`; Otherwise, `bp2` will be a null pointer
assert(bp2);
```

For more about dynamic cast, see [here](#). Notice that we should make sure the classes have one or more virtual methods.

## Lecture 22: Operator Overload

---

For full explanations see [Operators](#).

C++ lets us redefine the meaning of the operators when applied to objects of class type. This is known as operator overloading.

Like any other function, an overloaded operator has a return type and a parameter list.

### Unary operators

```
A A::operator-() const;
A& A::operator++();    // ++a
A A::operator++(int);  // a++
A& A::operator--();    // --a
A A::operator--(int);  // a--
```

### Binary operators

```
A& A::operator= (const A& rhs);
A& A::operator+= (const A& rhs);
A& A::operator-= (const A& rhs);
A operator+ (const A& lhs, const A& rhs);
istream& operator>> (istream& is, A& rhs);
ostream& operator<< (ostream& os, const A& rhs);
bool operator== (const A& lhs, const A& rhs);
bool operator!= (const A& lhs, const A& rhs);
bool operator< (const A& lhs, const A& rhs);
bool operator<= (const A& lhs, const A& rhs);
bool operator> (const A& lhs, const A& rhs);
bool operator>= (const A& lhs, const A& rhs);

// Especially:
const T& A::operator[] (size_t pos) const;
T& A::operator[] (size_t pos);
```

## Lecture 23: Linear List; Stack

---

### 1. Linear List

The main idea thing is that the operations are based by position. And if the position that you are going to access are invalid, you should throw **BoundsError** exception.

## 2. Stack

Property: last in, first out (*LIFO*).

### Methods of Stack

- **size()**: number of elements in the stack.
- **isEmpty()**: check if stack has no elements.
- **push(Object o)**: add object o to the **top** of the stack.
- **pop()**: remove the **top** object if stack is not empty; otherwise, throw **stackEmpty**.
- **Object &top()**: returns a reference to the top element.

### Implementation

#### Using Arrays

Need an **int size** to record the size of the stack.

- **size()**: return **size**;
- **isEmpty()**: return (**size == 0**);
- **push(Object o)**: add object o to the **top** of the stack and increment **size**. Allocate more space if necessary.
- **pop()**: If **isEmpty()**, throw **stackEmpty**; otherwise, decrement **size**.
- **Object &top()**: returns a reference to the top element **Array[size-1]**.

#### Using Linked Lists

- **size()**: **LinkedList::size()**;
- **isEmpty()**: **LinkedList::isEmpty()**;
- **push(Object o)**: insert object at the beginning **LinkedList::insertFirst(Object o)**;
- **pop()**: remove the first node **LinkedList::removeFirst()**;
- **Object &top()**: returns a reference to the object stored in the first node.

### Comparison: memory and time trade-off

1. Linked List:
  - memory efficient: a new item just needs extra constant amount of memory
  - time inefficient for **size()** operation because need to traverse the whole list
2. Array:
  - time efficient for **size()**.
  - memory inefficient: need to allocate a big enough array

### Applications

1. Function calls in C++
2. Web browser's "back" feature

### 3. Parentheses Matching

## Lecture 24: Queue

---

Very commonly used data structure. Basic principle: "First In First Out" (**FIFO**).

### Methods of Queue

Most important methods (almost all kinds of queues has these methods):

- **enqueue(Object o)**: add object o to the rear of the queue. (Usually returns nothing.)
- **dequeue()**: remove the front object of the queue if not empty; otherwise, throw an exception. (Usually returns the object that is removed.)
- **size()**: returns number of elements in the queue.
- **isEmpty()**: check if queue has no elements.

Other possible methods:

- **Object &front()**: returns a reference to the front element of the queue.
- **Object &rear()**: returns a reference to the rear element of the queue.

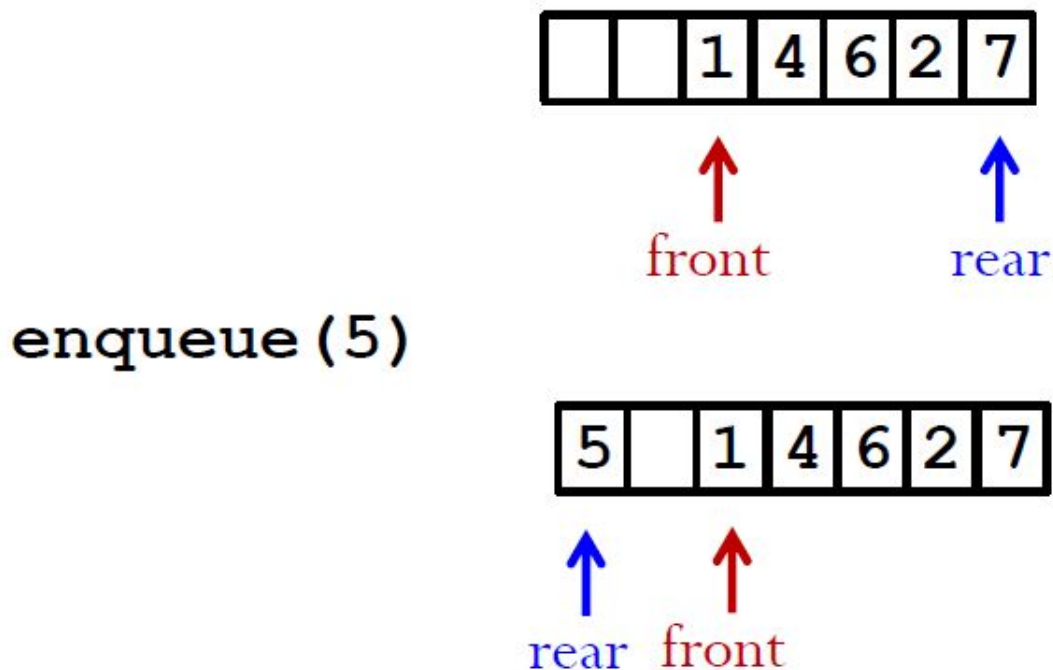
### Queues Using Linked Lists

How to implement the methods using linked lists:

- **enqueue(Object o)**: add a node to the end of the linked list.
- **dequeue()**: remove a node from the head of the linked list.
- **size()**: can iterate through the linked list and count the number of nodes.
- **isEmpty()**: check if the pointer to the linked list is **NULL**.
- **Object &front()**: returns a reference to the node at the head of the linked list.
- **Object &rear()**: returns a reference to the node at the end of the linked list.

### Queues Using Arrays

- Let the elements "drift" within the array.
- Maintain two integers to indicate the front and the rear of the queue (advance **front** when dequeuing; advance **rear** when inserting).
- Use a circular array (more space efficient):

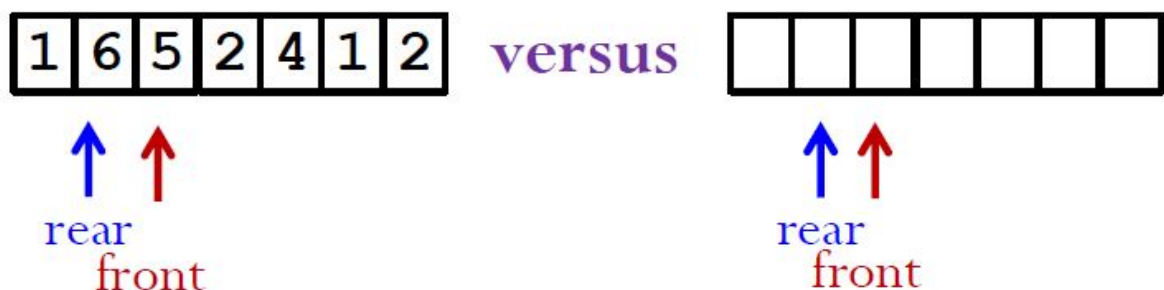


When inserting a new element, advance **rear** circularly; when popping out an element, advance **front** circularly.

Can be realized by

```
front = (front+1) % MAXSIZE;
rear = (rear+1) % MAXSIZE;
```

- Solve the problem of distinguishing an empty queue and full queue:



Maintain a **flag** indicating empty or full, or a **count** on the number of elements in the queue.

We can see that using array can be more complicated than linked list, and the size can be restricted by **MAXSIZE** of array. However, it can be easier to access the elements by index. For example, the index of the second element can be **front+1** (if not exceeding the capacity of array). Still, accessing elements by index is not very useful in queues.

## Deque

Property: Items can be inserted and removed from both ends of the list.

Methods:

- `push_front(Object o)`
- `push_back(Object o)`
- `pop_front()`
- `pop_back()`

The implementation can be more complicated than queue. Only use it if inserting and removing from both ends are truly necessary.

## Lecture 25, 26: STL

---

Three kinds of containers:

- **Sequential Containers:** let the programmer control the order in which the elements are stored and accessed. The order doesn't depend on the values of the elements.
- **Associative Containers:** store elements based on their *values*.
- **Container Adaptors:** take an existing container type and make it act like a different type

### Sequential Containers:

1. **vector:** based on arrays.
  - fast random access
  - fast insert/delete at the back
  - inserting / deleting at other position is slow
2. **deque** (double-ended queue): based on arrays.
  - fast random access (`operator[]`)
  - fast insert/delete at front or back
3. **list:** based on doubly-linked lists.
  - only bidirectional sequential access (no `operator[]`)
  - fast insert/delete at any point in the list

### I. Vector

1. How to use it

```
#include <vector>
using namespace std;

vector<int> ivec; // holds ints
vector<IntSet> isvec; // holds IntSets
```

1. **Initialization**

```
vector<T> v1; // empty vector v1
vector<T> v2(v1); // copy constructor
vector<T> v3(n,t); // construct v3 that has n elements with value t
```

### 1. Size

- `v.size()` returns a value of `size_type` corresponding to the vector type.
- `vector<int>::size_type`
  - a companion type of vector (why: to make the type machine-independent)
  - essentially an unsigned type
    - you can convert it into `unsigned int` but not `int`:

```
unsigned int s = v.size();
```

- not `vector::size_type`
- check whether empty: `v.empty()`

### 1. Add/Remove

- `v.push_back(t)`: add element `t` to the end of `v`
  - container elements are copies: no relationship between the element in the container and the value from which it was copied
- `v.pop_back()`: remove the last element in `v`. `v` must be non-empty

### 1. Subscripting Vector

```
vector<int>::size_type n;
v[n] = 0; //must ensure that the v.size() > n

for (n = 0; n != v.size(); ++n) {
    v[n] = 0;
}
```

### 1. other operations:

```
v1 = v2;
v.clear();
v.front(); //non-empty
v.back(); //non-empty
```



## Iterator

All of the library containers define iterator types, but only a few of them support subscripting.

1. Declaratio: `vector<int>::iterator it;`
2.
  - `v.begin()` returns an iterator pointing to the first element of vector
  - `v.end()` returns an iterator positioning to **one-past-the-end** of the vector
    - usually used to indicate when we have processed all the elements in the vector
    - If the vector is empty, the iterator returned by `begin` is the same as the iterator returned by `end`, e.g.
3. Operations:
  - dereference: can read/write through `*iter` (cannot dereference the iterator returned by `end()`)
  - `++iter`, `iter++`: next item (cannot increment the iterator returned by `end()`);  
`--iter`, `iter--` go back to the previous item
  - `iter == iter1` and `iter != iter1`: check whether two iterators point to the same data item

```
vector<int>::iterator begin = ivec.begin();
vector<int>::iterator end = ivec.end();
while (begin != end) {
    cout << *begin++ << " ";
    // 1. get the value of *begin
    // 2. cout << *begin << " ";
    // 3. begin++;
}
```

4. `const_iterator`: cannot change the value of the element it refers to.
  - Declaration: `vector<int>::const_iterator it;`
  - check `cbegin()` and `cend()`, which will return const iterator.

### 5. Iterator Arithmetic

**Not all** containers support! Check random access iterator [here](#) for example.

- `iter+n`, `iter-n`, where `n` is an integer
- example: go to the middle

```
vector<int>::iterator mid;
mid = v.begin() + v.size()/2;
```

6. Relational Operation: `>`, `>=`, `<`, `<=`

### Not all containers support!

- To compare, iterators must refer to elements in the same container or one past the end of the container (`c.end()`)

7. More about initialization of vector `vector<T> v(b,e)`: create vector `v` with a copy of the elements from the range denoted by iterators `b` and `e`.

- `[b,e)`
- check the constructor of the containers, `vector` for example.

```
vector<int> front(v.begin(), mid);
```

- You can use deque to initialize vector, e.g.

```
deque<string> ds(10, "abc");
vector<string> vs(ds.begin(), ds.end());
```

- Use array to initialize vector:

```
int a[] = {1, 2, 3, 4};
unsigned int sz = sizeof(a) / sizeof(int);
vector<int> vi(a, a+sz);
```

8. `v.insert(p, t)`

- inserts element with value `t` **right before** the element referred to by iterator `p`.
- returns an iterator referring to the element that was added

9. `v.erase(p)`

- removes element referred to by iterator `p`.
- returns an iterator referring to the element after the one deleted, or an **off-the-end** iterator if `p` referred to the last element.

## II. Deque

1. `#include <deque>`

2. Similarities with vector:

- **Initialization:**
  - `deque<T> d; deque<T> d(d1);`
  - `deque<T> d(n,t)`: create `d` with `n` elements, each with value `t`.
  - `deque<T> d(b,e)`: create `d` with a copy of the elements from the range denoted by iterators `b` and `e`.
- `size()`, `empty()`

- `push_back()`, `pop_back()`
- random access through `d[k]`
- `begin()`, `end()`, `insert(p,t)`, `erase(p)`
- operations on iterators: `*iter`, `++iter`, `--iter`, `iter != iter2`, `iter == iter2`.....

### 3. Difference with vector:

- `d.push_front(t)`
- `d.pop_front(t)`

## III. List

### 1. `#include <list>`

### 2. only bidirectional sequential access

### 3. Similarities with vector:

- Initialization:
  - `list<T> l; list<T> l(l1);`
  - `list<T> l(n,t)`: create `l` with `n` elements, each with value `t`.
  - `list<T> l(b,e)`: create `l` with a copy of the elements from the range denoted by iterators `b` and `e`.
- `size()`, `empty()`
- `push_back()`, `pop_back()`
- `begin()`, `end()`, `insert(p,t)`, `erase(p)`
- operations on iterators: `*iter`, `++iter`, `--iter`, `iter != iter2`, `iter == iter2`.....

### 4. Difference with vectors

- not support subscripting
- not support iterator arithmetic, i.e. cannot do `it+3`, you can only use `++/--`
- no relational operation `<`, `<=`, `>`, `>=`, you can only use `==` and `!=` to compare
- `l.push_front(t)`
- `l.pop_front(t)`

## Choose appropriate containers

1. Use vector, unless have other good reasons
2. require random access: vector or deque
3. insert or delete elements in the middle: list
4. insert or delete elements at the front and the back: deque
5. others: check the predominant operations

## Container Adaptors

e.g. `stack`, `queue`, `priority_queue` (will be learned in 281)

1. `#include <stack>` and `#include <queue>`
2. By default, there are implemented using deque. So if you initialize it in this way: `stack<int> stk(deq);`, `deq` must be a deque. But you can also use vector or list to build stack:

```
// Assume ivec is vector<int>
stack<int, vector<int> > stk(ivec);
//                ^ theres is a space here
```

1. Notice that for queue, you can use **deque and list to build it (no vector here)**. Check the link.
2. Be aware that the names of the operations are different with previous ones. Check the link.

stack	queue
s.empty()	q.empty()
s.size()	q.size()
s.pop()	q.pop()
s.push(item)	q.push(item)
<b>s.top()</b>	<b>q.front()</b>

## Associative Container

### ADT: Dictionary

1. quickly add (key, value) pair and retrieve value by key
2. Methods
  - Value find(Key K): return the value whose key is K. Return Null if none.
  - void insert(Key K, Value V): Insert a pair (K, V) into the dictionary. If the pair with key as K already exists, **update** the value.
  - Value remove(Key K): Remove the pair with key as K from the dictionary and return its value. Return Null if none.
  - int size(): return number of pairs in the dictionary.
3. Implementation: array or linked list.

### Applications:

1. map: dictionary, elements are (key, value) pairs
2. set: contains only a key and supports efficient queries to whether a given key is present.

### Map: associative array

Important: key cannot be duplicate

1. `#include <map>`
2. **Constructors:**
  - `map<k, v> m;` // create an empty map named m
  - `map<k, v> m(m1);` // create m as a copy of m1, the two maps must have the same key and value types

- `map<k, v> m(b, e)` // create m as a copy of the elements from the range denoted by iterators b and e

3. Elements are sorted by keys, so the key type should have: strict weak ordering (<)

4. iterators: `map<string, int>::iterator it = word_count.begin()`, then `*it` is a reference to a `pair<const string, int>`.

- Please notice that `it->first` is const, so we cannot write `it->first = "new key";`.
- To access the value, use `it->second`, you can assign new value to it.
- Note that here we are using arrows (->) to access its key or value.

5. Adding elements to a map

- use subscript operator (different with `vector`)
  - if the key is already there in the map, then there is nothing new. You can read or update the value using `m[k]=v`.
  - if the key is not there, `m[k] = v` will
    1. search the key `k` and don't find it.
    2. insert (k, default\_v) to the map. If `v` is `int`, it will insert (k, 0).
    3. update (k, default\_v) to (k, v)
- use `insert` member:
  - `m.insert(e)`, where `e` is a (key, value) pair. If the key is not in m, then insert the pair. Otherwise, does nothing.
  - returns a **pair** of (map iterator, bool)
    - map iterator refers to the element with key
    - bool indicates whether the element was inserted or not

```
#include <iostream>
#include <map>

int main (){
    std::map<char,int> mymap;

    // first insert function version (single parameter):
    mymap.insert ( std::pair<char,int>('a',100) );
    // can also use mymap.insert(std::make_pair('a', 100))
    // remember to #include <utility> if you use make_pair
    mymap.insert ( std::pair<char,int>('z',200) );

    std::pair<std::map<char,int>::iterator,bool> ret;
    ret = mymap.insert ( std::pair<char,int>('z',500) );
    if (ret.second==false) {
        std::cout << "element 'z' already existed";
        std::cout << " with a value of " << ret.first->second <<
'\n';
    }
}
```

6. find elements: `m.find(k)`

- returns an iterator to the element indexed by key `k` if there is one
- else, returns an off-the-end iterator: compare with `end()` to check whether the key is there

```
int occurs = 0;
map<string, int>::iterator it = world_count.find("abc");
if (it != world_count.end()) occurs = it->second;
```

## 7. remove elements:

- `m.erase(iter)`
  - removes element referred to by the iterator `iter` from `m`. `iter` must refer to an actual element in `m`; it must not be `m.end()`
  - returns void
- `m.erase(k)`
  - removes the element with key `k` from `m` if it exists
  - otherwise, do nothing
  - returns the number of elements removed, either 0 or 1 for map.

8. iterate across a map: When we use a niterator to traverse a map, the iterators yield elements in ascending key order.

```
map<string, int>::iterator it;
for(it = word_count.begin(); it != word_count.end(); ++it) {
    cout << it->first << " occurs" << it->second << " times";
}
```

For the STL part, please search the thing you need in [www.cplusplus.com/reference/](http://www.cplusplus.com/reference/) or [en.cppreference.com/w/](http://en.cppreference.com/w/) often