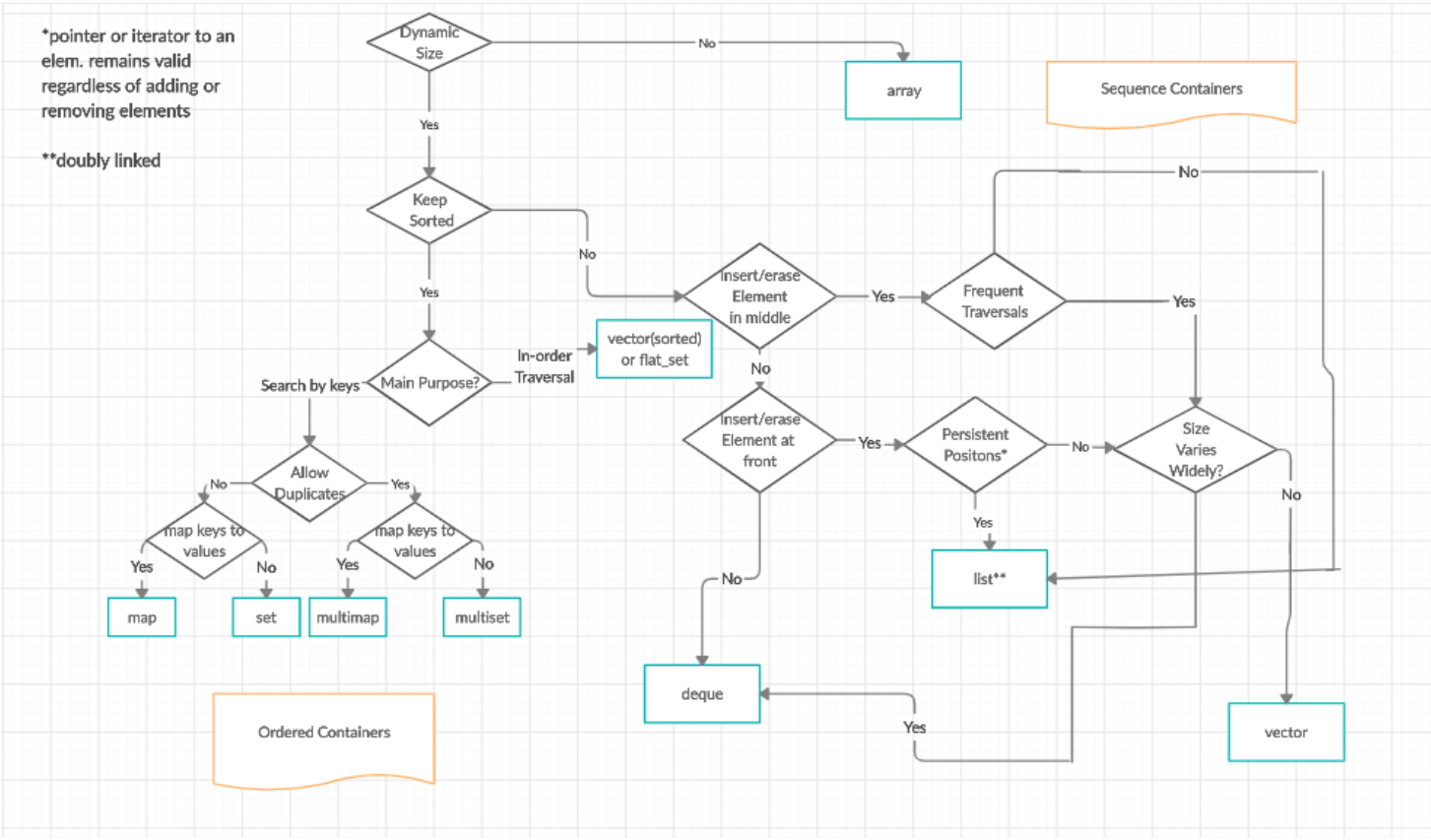
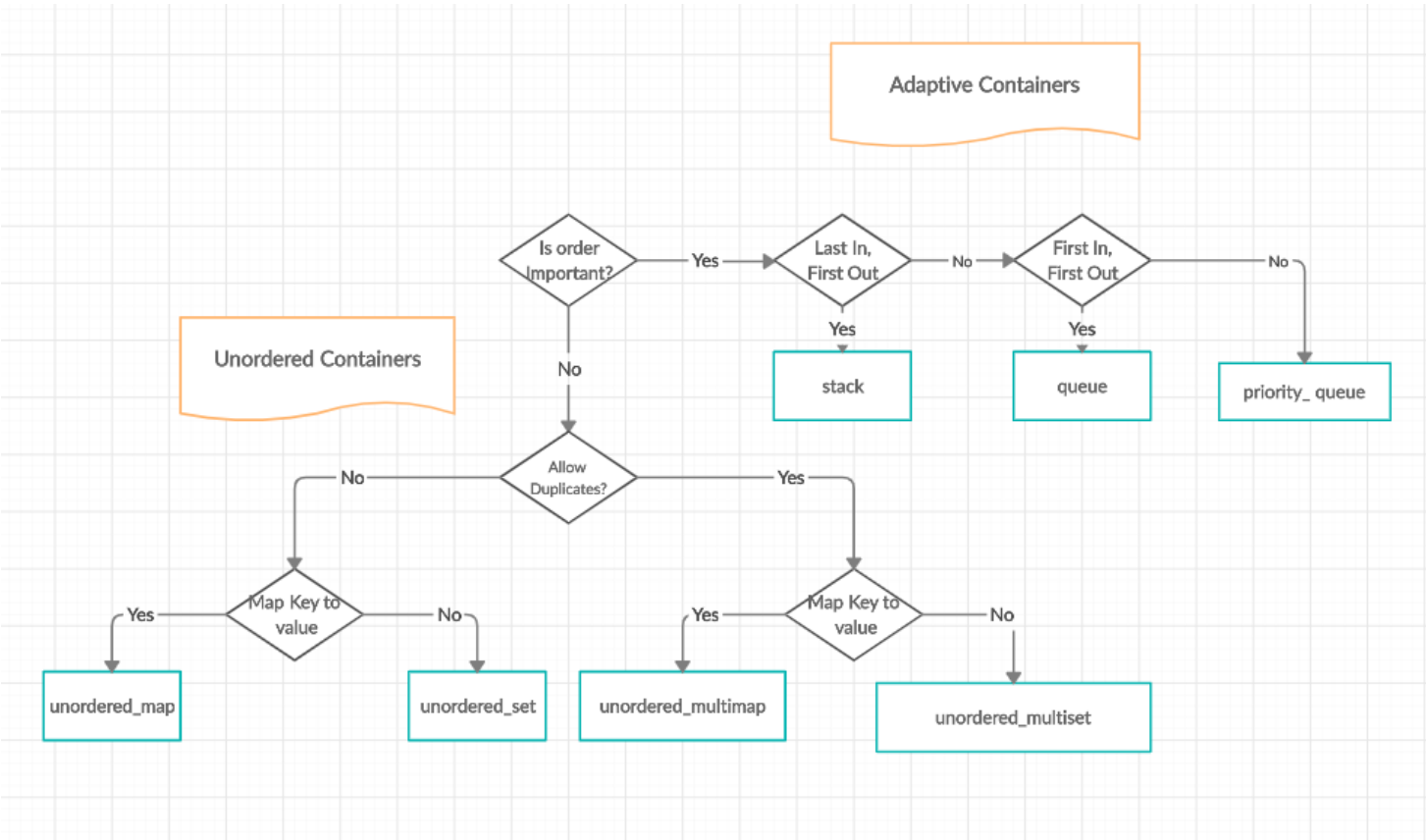
总述


*pointer or iterator to an elem. remains valid regardless of adding or removing elements
**doubly linked
Dynamic Size
No
array
Sequence Containers
Yes
Keep Sorted
No
Yes
Insert/erase Element in middle
Yes
Frequent Traversals
Yes
No
No
vector(sorted) or flat_set
In-order Traversal
Main Purpose?
Search by keys
Insert/erase Element at front
Yes
Persistent Positons*
No
Size Varies Widely?
No
Yes
Allow Duplicates
No
Yes
map keys to values
map keys to values
Yes
No
Yes
No
map
set
multimap
multiset
list**
No
deque
Yes
vector
Ordered Containers


Adaptive Containers
Is order Important?
Yes
Last In, First Out
No
First In, First Out
No
Yes
Yes
stack
queue
priority_ queue
Unordered Containers
No
Allow Duplicates?
No
Yes
Map Key to value
Map Key to value
Yes
No
Yes
No
unordered_map
unordered_set
unordered_multimap
unordered_multiset

字符串string

C++ 在STL中定义了string来方便使用字符串

string 封装了很多实用的成员方法,包括：查找find，拷贝copy，删除erase，替换replace，插入insert等
同时string类不需要考虑内存释放和越界，本质上是一个动态的char数组

基本操作

首先是构造string

```
string();
// 默认构造函数，创建一个空的字符串

string(const string& str);
// 拷贝构造函数，使用一个string对象初始化另一个string对象

string(const char* s);
// 含参构造函数，使用C风格字符串初始化

string(int n, char c);
// p含参构造函数，使用n个字符c初始化
```

string接受各种赋值方式，包括采用c风格，使用另一个string，使用字符赋值等等，可以直接使用 = 进行赋值
同时，也可以使用 `assign()` 成员函数进行赋值，如：

```
// string&表示该方法返回一个string对象
string& assign(const char* s);
// C风格字符串赋值给当前的字符串
string& assign(const char* s, int n);
// 把C风格字符串s的前n个字符赋给当前的字符串
string& assign(const string& s);
// 把字符串s赋给当前字符串
string& assign(int n, char c);
// 把n个字符c赋给当前的字符串
string& assign(const string& s, int start, int n);
// 将字符串s中从start开始的n个字符赋值给当前字符串
```

我们知道，string是由char组成的。和C类似，可以使用 `[n]` 来获取操作符，使用下标操作符获取字符时，如果下标越界，程序将会强制终止
亦可以使用 `at` 还是来获取字符
使用at方法获取字符时，如果下标越界，at方法内部会抛出异常（exception），可以使用try-catch捕获并处理该异常。示例如下：

```
#include <stdexception>
//标准异常头文件
#incldue <iostream>
using namespace std;

int main()
{
    string s = "hello world";
    try
    {
        //s[100]不会抛出异常，程序会直接挂掉
        s.at(100);
    }
    catch (out_of_range& e)
        //如果不熟悉异常类型，可以使用多态特性， catch(exception& e)。
    {
        cout << e.what() << endl;
        //打印异常信息
    }
    return 0;
}
```

string同样提供了两种办法来进行字符串拼接，首先是 `+=`：

```
string& operator+=(const string& str);
// 将字符串str追加到当前字符串末尾
string& operator+=(const char* str);
// 将C风格字符数组追加到当前字符串末尾
string& operator+=(const char c);
// 将字符c追加到当前字符串末尾
/* 上述操作重载了复合操作符+= */
```

也可以使用 `append()` 方法进行拼接：

```
string& append(const char* s);
// 把C风格字符数组s连接到当前字符串结尾
string& append(const char* s, int n);
// 把C风格字符数组s的前n个字符连接到当前字符串结尾
string& append(const string &s);
// 将字符串s追加到当前字符串末尾
string& append(const string&s, int pos, int n);
// 把字符串s中从pos开始的n个字符连接到当前字符串结尾
string& append(int n, char c);
// 在当前字符串结尾添加n个字符c
```

对字符串操作

首先是查找和替换功能
使用 `find()` 方法来进行查找,查找包括字符串查找和字符查找
find默认查找第一次出现的位置，可以指定范围：

```
int find(const string& str, int pos = 0) const;
// 查找str在当前字符串中第一次出现的位置，从pos开始查找，pos默认为0
int find(const char* s, int n = 0) const;
// 查找C风格字符串s在当前字符串中第一次出现的位置，从pos开始查找，pos默认为0
int find(const char* s, int pos, int n) const;
// 从pos位置查找s的前n个字符在当前字符串中第一次出现的位置
int find(const char c, int pos = 0) const;
// 查找字符c第一次出现的位置，从pos开始查找，pos默认为0
```

```
//当查找失败时，find方法会返回-1，-1已经被封装为string的静态成员常量string::npos。
static const size_t nops = -1;
```

使用 `rfind()` 来查找某个字符最后一次出现的位置，使用方法类似find：

```
int rfind(const string& str, int pos = npos) const;
// 从pos开始向左查找最后一次出现的位置，pos默认为npos
int rfind(const char* s, int pos = npos) const;
// 查找s最后一次出现的位置，从pos开始向左查找，pos默认为npos
int rfind(const char* s, int pos, int n) const;
// 从pos开始向左查找s的前n个字符最后一次出现的位置
int rfind(const char c, int pos = npos) const;
// 查找字符c最后一次出现的位置
```

> find方法通常查找字串第一次出现的位置，而rfind方法通常查找字串最后一次出现的位置。rfind(str, pos)的实际的开始位置是pos + str.size()，即从该位置开始（不包括该位置字符）向前寻找匹配项，如果有则返回字符串位置，如果没有返回string::npos。-1其实是size_t类的最大值（学过补码的同学应该不难理解），所以string::npos还可以表示“直到字符串结束”，可以以此来理解rfind中pos的默认参数。

使用 `replace()` 方法来进行替换：

```
string& replace(int pos, int n, const string& str);
// 替换从pos开始n个字符为字符串s
string& replace(int pos, int n, const char* s);
// 替换从pos开始的n个字符为字符串s
```

对于string的相互比较，使用 `compare()` 方法，compare函数依据字典序比较，在当前字符串比给定字符串小时返回-1，在当前字符串比给定字符串大时返回1，相等时返回0

```
int compare(const string& s) const; // 与字符串s比较
int compare(const char* s) const; // 与C风格字符数组比较
```

更进一步的，string类重载了所以比较操作符，可以直接使用操作符进行比较的判断，含义同compare:

```
bool operator<(const string& str) const;
bool operator<(const char* str) const;
bool operator<=(const string& str) const;
bool operator<=(const char* str) const;
bool operator==(const string& str) const;
bool operator==(const char* str) const;
bool operator>(const string& str) const;
bool operator>(const char* str) const;
bool operator>=(const string& str) const;
bool operator>=(const char* str) const;
bool operator!=(const string& str) const;
bool operator!=(const char* str) const;
```

使用 `substr()` 方法来得到子串：

```
string substr(int pos = 0, int n = npos) const;
// 返回由pos开始的n个字符组成的字符串
```

使用 `insert()` 来进行插入，使用 `erase()` 进行擦除：

```
string& insert(int pos, const char* s); // 在pos位置插入C风格字符数组
string& insert(int pos, const string& str); // 在pos位置插入字符串str
string& insert(int pos, int n, char c); // 在pos位置插入n个字符c

// 返回值是插入后的字符串结果，erase同理。其实就是指向自身的一个引用。
string& erase(int pos, int n = npos); // 删除从pos位置开始的n个字符
```

其它

有时候需要进行 `string` 和 `c-Style string` 的转化

```
// string -> c style
string str = "demo";
const char* cstr = str.c_str(); //使用c_str()方法

// c style -> string
const char* cstr = "demo";
string str(cstr); // 本质上其实是一个有参构造，见构造方法
```

> 在c++中存在一个从 const char* 到 string 类的隐式类型转换，但却不存在从一个 string 对象到 const char* 的自动类型转换。对于 string 类型的字符串，可以通 过c_str() 方法返回 string 对象对应的 const char* 字符数组。比如说，当一个函数的参数是 string 时，我们可以传入 const char* 作为参数，编译器会自动将其转化为 string ，但这个过程不可逆。为了修改 string 字符串的内容，下标操作符 [] 和 at 都会返回字符串的引用，但当字符串的内存被重新分配之后，可能发生错误。（结合字符串的本质是动态字符数组的封装便不难理解了）

以及有部分和string相关的全局函数:
大小写转换:
单个字符:

```
#include <cctype>
// 在iostream中已经包含了这个头文件，如果没有包含iostream头文件，则需手动包含cctype

int tolower(int c); // 如果字符c是大写字母，则返回其小写形式，否则返回本身
int toupper(int c); // 如果字符c是小写字母，则返回其大写形式，否则返回本身

/**
  * C语言中字符就是整数，这两个函数是从C库沿袭过来的，保留了C的风格
*/
```

如果想要对整个字符串进行大小写转化，则需要使用一个for循环，或者配合和algorithm库来实现。例如

```
#include <string>
#include <cctype>
#include <algorithm>

string str = "Hello, World!";
transform(str.begin(), str.end(), str.begin(), toupper); //字符串转大写
transform(str.begin(), str.end(), str.begin(), tolower); //字符串转小写
```

字符串和数字转化:

```
// ============================
// 数转字符串
// c++11标准新增了全局函数std::to_string，十分强大，可以将很多类型变成string类型
#include <string>
using namespace std;

/** 带符号整数转换成字符串 */
string to_string(int val);
string to_string(long val);
string to_string(long long val);

/** 无符号整数转换成字符串 */
string to_string(unsigned val);
string to_string(unsigned long val);
string to_string(unsigned long long val);

/** 实数转换成字符串 */
string to_string(float val);
string to_string(double val);
string to_string(long double val);

// ============================
// 字符串转数
#include <cstdlib>
#include <string>
using namespace std;

/** 字符串转带符号整数 */
int stoi(const string& str, size_t* idx = 0, int base = 10);
long stol(const string& str, size_t* idx = 0, int base = 10);
long long stoll(const string& str, size_t* idx = 0, int base = 10);

/**
  * 1. idx返回字符串中第一个非数字的位置，即数值部分的结束位置
  * 2. base为进制
  * 3. 该组函数会自动保留负号和自动去掉前导0
 */

/** 字符串转无符号整数 */
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);

/** 字符串转实数 */
float stof(const string& str, size_t* idx = 0);
double stod(const string& str, size_t* idx = 0);
long double stold(const string& str, size_t* idx = 0);
```

与之类似的在同一个库里的还有一组**基于字符数组**的函数如下：

```
// 'a' means array, since it is array-based.

int atoi(const char* str); // 'i' means  int
long atol(const char* str); // 'l' means long
long long atoll(const char* str); // 'll' means long long

double atof(const char* str); // 'f' means double
```

向量_队列_栈

Vector

vector的数据安排及操作方式，与array非常相似，两者的唯一差别在于空间的运用的灵活性,array是静态空间，vector是动态空间并且配置空间的策略也考虑了运行成本，采用特定的扩展的策略

迭代器遍历

```
// 使用迭代器进行正序遍历
for (vector<T>::iterator it = v.begin(); it != v.end(); it++)
{
    cout << *it << endl;
}
/**
 * 1. 迭代器的声明方式：  容器类型::迭代器类型
 * 2. 顺序首尾迭代器由begin()和end()方法生成
*/

// 使用迭代器逆序遍历
for (vector<T>::reverse_iterator it = v.rbegin(); it != v.rend(); it++)
{
    cout << *it << endl;
}
/**
  * 1. 逆向迭代器不再是iterator，而是reverse_iterator
  * 2. 逆序首位迭代器由rbegin()和rend()方法生成
*/
```

此外，介绍一种现场测试迭代器是否能随机访问的方法:

```
iterator++;
iterator--;
//通过编译，至少是双向迭代器

iterator = iterator + 1;
//通过编译，则是随机访问迭代器
```

vector采用的数据结构非常简单，线性连续空间，它以两个迭代器 `_Myfirst` 和 `_Mylast` 分别指向配置得来的连续空间中已被使用的范围，并以迭代器 `Myend` 指向整块连续内存空间的尾端

> 所谓动态增加大小，并不是在原空间之后续接新空间（因为无法保证原空间之后尚有可配置的空间），而是一块更大的内存空间，然后将原数据拷贝新空间，并释放原空间。

> 因此，对vector的任何操作，一旦引起空间的重新配置，指向原vector的所有迭代器就都失效了

vector常用API操作

构造函数：

```
vector<T> v; // 采用模版类实现，默认构造函数
vector<T> v(T* v1.begin(), T* v1.end()); // 将v1[begin(), end())区间中的元素拷贝给本身
vector<T> v(int n, T elem); // 将n个elem拷贝给本身
vector<T> v(const vector<T> v1); // 拷贝构造函数

// 下面对于第二种构造方式给出一个特殊的例子：
int array[5] = {1, 2, 3, 4, 5};
vector<int> v(array, array + sizeof(array) / sizeof(int));
// 联系我们之前提到的vector迭代器本质上是指针就不难理解了
```

赋值：

```
assign(beg, end); // 将[beg, end)区间中的数据拷贝复制给本身
assign(n, elem); // 将n个elem拷贝给本身
vector& operator=(const vector& vec); // 重载赋值操作符

// 互换操作也可视为一种特殊的赋值：
swap(vec); //将vec与本身的元素互换

// 巧用swap来收缩空间：
vector<int>(v).swap(v);
// vector<int>(v)：利用拷贝构造函数初始化匿名对象
// swap(v)：交换的本质其实只是互换指向内存的指针
// 匿名对象指针指向的内存会由系统自动释放
```

大小操作系列：

```
int size(); // 返回容器中的元素个数
bool empty(); // 判断容器是否为空

void resize(int num);
// 重新指定容器的长度为num，若容器变长，则以默认值填充新位置。
// 若容器变短，则末尾超出容器长度的元素被删除
void resize(int num, T elem);
// 重新指定容器的长度为num，若容器变长，则以elem填充新位置。
// 若容器变短，则末尾超出容器长度的元素被删除

int capacity(); // 返回容器的容量
void reserve(int len);
// 容器预留len个元素长度，预留位置不初始化，元素不可访问
```

数据存取操作：

```
T& at(int idx); // 返回索引idx所指的数据，如果idx越界，抛出out_of_range异常
T& operator[](int idx); // 返回索引idx所指的数据，如果idx越界，运行直接报错

T& front(); // 返回首元素的引用
T& back(); // 返回尾元素的引用
```

插入和删除操作：

```
insert(const_iterator pos, T elem); // 在pos位置处插入元素elem
insert(const_iterator pos, int n, T elem); // 在pos位置插入n个元素elem
insert(pos, beg, end); // 将[beg, end)区间内的元素插到位置pos
push_back(T elem); // 尾部插入元素elem
pop_back(); // 删除最后一个元素

erase(const_iterator start, const_iterator end); // 删除区间[start, end)内的元素
erase(const_iterator pos); // 删除位置pos的元素

clear(); // 删除容器中的所有元素

//
std::vector<std::wstring> v2(3, L"c");
v2.insert(v2.begin()+4, L"3");   //在指定位置，例如在第五个元素前插入一个元素

v2.insert(v2.end(), L"3");   //在末尾插入一个元素

v2.push_back(L"9");   //在末尾插入一个元素

v2.insert(v2.begin(), L"3");   //在开头插入一个元素
```

Deque , Queue

vector 容器是单向开口的连续内存空间，deque 则是一种双向开口的连续线性空间
二者的主要差异体现在：
deque 允许使用常数项时间在头部插入或删除元素
deque 没有容量的概念，因为它是由动态的分段连续空间组合而成，随时可以增加一块新的空间并链接起来

Deque

遍历：

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

const deque<T> d;
for (deque<T>::const_iterator it = d.begin(); it != d.end(); it++)
// 要用const_iterator指向常量容器
{
        // 如果在此处修改it指向空间的值，编译器会报错
        cout << *it << endl;
}

/**
  * iterator 普通迭代器
  * reverse_iterator 反转迭代器
  * const_iteratoe 只读迭代器
*/
```

常用API：
构造函数：

```
deque<T> deqT; // 默认构造函数
deque(beg, end); // 构造函数将[beg, end)区间中的元素拷贝给本身
deque(int n, T elem); // 构造函数将n个elem拷贝给本身
deque(const deque& deq); // 拷贝构造函数
```

赋值操作：

```
assign(beg, end); // 将[beg, end)区间中的元素拷贝赋值给本身
assign(int n, T elem); // 将n个元素elem拷贝赋值给本身

deque& operator=(const deque& deq); // 重载赋值操作符

swap(deq); // 将deq与本身的元素互换
```

大小操作:

```
int size(); // 返回容器中元素的个数
bool empty(); // 判断容器是否为空

void resize(int num);
// 重新指定容器的长度为num，若容器变长，则以默认值填充新位置，
// 如果容器变短，则末尾超出容器长度的元素被删除
void resize(int num, T elem);
// 重新指定容器的长度为num，若容器变长，则以elem填充新位置，
// 如果容器变短，则末尾超出容器长度的元素被删除
```

插入和删除,存读:

```
// push是加pop是弹，_后跟前或者后
push_back(T elem); // 在容器尾部添加一个元素
push_front(T elem); // 在容器头部插入一个元素

pop_back(); // 删除容器最后一个数据
pop_front(); // 删除容器第一个数据

const_iterator insert(const_iterator pos, T elem);
// 在pos位置处插入元素elem的拷贝，返回新数据的位置
void insert(const_iterator pos, int n, T elem);
// 在pos位置插入n个元素elem，无返回值
void insert(pos, beg, end);
// 将[beg, end)区间内的元素插到位置pos，无返回值

clear(); // 移除容器的所有数据
iterator erase(iterator beg, iterator end);
// 删除区间[beg, end)的数据，返回下一个数据的位置
iterator erase(iterator pos);
// 删除pos位置的数据，返回下一个数据的位置

T& at(int idx); // 返回索引idx所指的数据，如果idx越界，抛出out_of_range异常
T& operator[](int idx); // 返回索引idx所指的数据，如果idx越界，运行直接报错

T& front(); // 返回首元素的引用
T& back(); // 返回尾元素的引用
```

Queue

queue 是一种先进先出(First In First Out, FIFO)的数据结构，它有两个出口，queue容器允许从一端新增元素，从另一端移除元素

queue 没有迭代器:
只有queue的顶端元素，才有机会被外界去用
queue不提供遍历功能，也不提供迭代器

常用API：

```
// 构造函数
queue<T> queT; // queue对象的默认构造函数形式，采用模版类实现
queue(const queue& que); // 拷贝构造函数

// 存取，插入，删除
void push(T elem); // 往队尾添加元素
void pop(); // 从队头移除第一个元素
T& back(); // 返回最后一个元素
T& front(); // 返回第一个元素

// 赋值
queue& operator=(const queue& que); // 重载赋值操作符

// 大小操作
bool empty(); // 判断队列是否为空
int size(); // 返回队列的大小
```

Stack

基本概念

stack 是一种**先进后出（First In Last Out, FILO）**的数据结构，它只有一个出口。
stack 容器允许新增元素、移除元素、取得栈顶元素，但是除了最顶端外，没有任何其他方法可以存取stack的其他元素。换言之，stack不允许有遍历行为

常用API：

```
stack<T> stkT; // 默认构造函数，stack采用模版类实现
stack(const stack& stk); // 拷贝构造函数

stack& operator=(const stack& stk); // 重载赋值操作符

void push(T elem); // 向栈顶添加元素
void pop(); // 从栈顶移除第一个元素
T& top(); // 返回栈顶元素

bool empty(); // 判断堆栈是否为空
int size(); // 返回栈的大小
```

> 注：在C++中，pop之负责弹出，使用top来得到目前的要的元素（无论是队列还是栈等）。因此在逐个取出（获得并删除）元素的时候，应该先用top再pop，或者使用迭代器。

链表_集合_映射

List

基本概念

链表是一种物理存储单元上非连、续非顺序的储存结构，数据元素的逻辑顺序是通过链表中的指针链接次序实现的
List由结点构成，而结点包括数据和下一个结点的信息，结点是在运行时动态生成的

> 它的好处是每次插入或者删除一个元素，就是配置或者释放一个元素的空间;因此，list对于空间的运用有绝对的精准，一点也不浪费;而且，list对于任何位置插入或删除元素都是常数项时间

在C++中，List是一个双向链表，灵活，但是空间和时间的额外消耗会比较大
list不仅仅是一个双向链表，而且是一个循环的双向链表

迭代器

首先是迭代器方面：

> ist 迭代器必须有能力指向list的结点，并有能力进行正确的递增、递减、取值、成员存取操作;由于list是一个双向链表，迭代器必须能够具备前移、后移的能力，所以list容器提供的是Bidirectional Iterators，双向迭代器;同时，list 有一个重要的性质，插入和删除操作都不会造成原有list迭代器的失效（这一点和vector不同，vector在进行插入的时候可能会造成内存的重新配置，导致原有的迭代器全部失效）

```
// 顺序遍历
for(list<T>::iterator it = lst.begin(); it != lst.end(); it++)
{
    cout << *it << endl;
}

// 逆序遍历
for(list<T>::reverse_iterator it = lst.rbegin(); it != lst.rend(); it++)
{
    cout << *it << endl;
}
```

常用API

构造函数：

```
list<T> lstT; // 默认构造形式，list采用模版类实现
list(beg, end); // 构造函数将[beg, end)区间内的元素拷贝给本身
list(int n, T elem); // 构造函数将n个elem拷贝给本身
list(const list& lst); // 拷贝构造函数
```

插入和删除操作：

```
void push_back(T elem); // 在容器尾部加入一个元素
void pop_back(); // 删除容器中最后一个元素

void push_front(T elem); // 在容器开头插入一个元素
void pop_front(); // 从容器开头移除第一个元素

insert(iterator pos, elem); // 在pos位置插入elem元素的拷贝，返回新数据的位置
insert(iterator pos, n, elem); // 在pos位置插入n个elem元素的拷贝，无返回值
insert(iterator pos, beg, end); // 在pos位置插入[beg, end)区间内的数据，无返回值

void clear(); // 移除容器的所有数据

erase(beg, end); // 删除[beg, end)区间内的所有数据，返回下一个数据的位置
erase(pos); // 删除pos位置的数据，返回下一个数据的位置

remove(elem); // 删除容器中所有与elem匹配的元素
```

大小操作和赋值操作：

```
int size(); // 返回容器中元素的个数
bool empty(); // 判断容器是否为空

void resize(int num);
// 重新制定容器的长度为num，若容器变长，则以默认值填充新位置；
// 若容器变短，则末尾超出容器长度的元素被删除
void resize(int num, T elem);
// 重新制定容器的长度为num，若容器变长，则以elem填充新位置；
// 若容器变短，则末尾超出容器长度的元素被删除
```

其它操作：

```
// 存取
T& front(); // 返回第一个元素
T& back(); // 返回最后一个元素

// 反转排序：
void reverse(); // 反转链表

void sort(); // 默认list排序，规则为从小到大
void sort(bool (*cmp)(T item1, T item2)); // 指定排序规则的list排序

// 不能用sort(lst.begin(), lst.end())
// 因为所有系统提供的某些算法（比如排序），其迭代器必须支持随机访问
// 不支持随机访问的迭代器的容器，容器本身会对应提供相应的算法的接口
```

Set / Multiset

基本概念和特性

set的特性是，所有的容器都会根据元素自身的键值进行自动被排序，不像map那样可以同时拥有实值和键值，set的元素既是实值又是键值

对于内部的数值，set不允许两个元素具有相同数值，也不允许通过其迭代器改变set元素的数值

> 换句话说，set的iterator是一种const_iterator

multiset特性及用法和set完全相同，唯一的差别在于它允许键值重复;set和multiset的底层实现是红黑树，红黑树为平衡二叉树的一种

基本操作：

构造和赋值：

```
set<T> st; // set 默认构造函数
multiset<T> mst; // multiset 默认构造函数
set(const set& st); // 拷贝构造函数

// ===
set& operator=(const set& st); // 重载等号操作符

swap(st); // 交换两个集合容器
```

大小操作和插入删除操作：

```
// 大小操作:
int size(); // 返回容器中元素的数目
bool empty(); // 判断容器是否为空

// 插入和删除操作
pair<iterator, bool> insert(T elem);
// 在容器中插入元素，返回插入位置的迭代器（不成功则返回end()）和是否插入成功
// 如果是multiset，则返回值只有iterator
clear(); // 清除所有元素
iterator erase(pos); // 删除pos迭代器所指的元素，返回下一个元素的迭代器
iterator erase(beg, end); // 删除区间[beg, end)内的所有元素，返回下一个元素的迭代器
erase(T elem); // 删除容器中值为elem的元素

//利用仿函数 指定set容器的排序规则
class MyCompare
{
public:
    bool operator()(int v1, int v2)
    {
        return v1 > v2;
    }
};

set<int, MyCompare> s;

//模版类也是可以有默认值的，第二个模版参数的默认值为less

// 同时，自定义的数据类型需要指出排序规则。当然，也可以通过重载小于操作符的方式指出
```

查找操作:

```
iterator find(T key);
// 查找键key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回set.end();
int count(T key);
// 查找键key的元素个数
iterator lower_bound(T keyElem);
// 返回第一个key>=keyElem元素的迭代器
iterator upper_bound(T keyElem);
// 返回第一个key>keyElem元素的迭代器
pair<iterator, iterator> equal_range(T keyElem);
// 返回容器中key与keyElem上相等的两个上下限迭代器

// 上述几个方法若不存在，返回值都是尾迭代器。
```

在C++的set中，使用find()方法得到一个迭代器后，可以通过解引用该迭代器来获取对应的数值。例如，如果你有一个名为myset的set，其中包含一些整数，你可以使用以下代码来查找值为5的元素并获取它的值

```
std::set<int> myset = {2, 4, 6, 8};
std::set<int>::iterator it = myset.find(5);
if (it != myset.end()) {
    std::cout << "Found " << *it << '\n';
} else {
    std::cout << "Not found\n";
}
```

对组的构造和使用：

```
//构造
pair<T1, T2> p(k, v);
//另一种构造方式
pair<T1, T2> p = make_pair(k, v);
//使用
cout << p.first << p.second << endl;
```

Map / Multimap

基本概念

map 的特性是，所有的元素都会根据元素的键值自动排序；
map 的所有元素都是pair，同时拥有实值和键值；pair的第一元素被视为键值，第二元素被视为实值；
map不允许两个元素有相同的键值；

和set类似的原因，我们不能通过迭代器改变map的键值，但我们可以任意修改实值。就是可以通过迭代器来修改value，但是不能修改key

map和multimap的操作类似，唯一的区别是multimap键值可重复；map和multimap都是以红黑树作为底层实现机制。

基本操作

遍历：

```
for (map<T1, T2>::iterator it = m.begin(); it != m.end(); it++)
{
    cout << "key = " << it->first << " value = " << it->second << endl;
}
```

常用API：

构造函数,赋值，大小操作：

```
map<T1, T2> mapTT; // map默认构造函数
map(const map& mp); // 拷贝构造函数

map& operator=(const map& mp); // 重载等号操作符
swap(mp); // 交换两个集合容器

int size(); // 返回容器中元素的数目
bool empty(); // 判断容器是否为空
```

插入，删除：

```
pair<iterator, bool> insert(pair<T1, T2> p); // 通过pair的方式插入对象
/*
1. 参数部分可以用pair的构造函数创建匿名对象
2. 也可以使用make_pair创建pair对象
3. 还可以用map<T1, T2>::value_type(key, value)来实现
*/

T2& operator[](T1 key); // 通过下标的方式插入值
// 如果通过下标访问新的键却没有赋值，会自动用默认值填充

// 另外，map指定排序规则的方式和set类似，都是利用functor在模版类型表的最后一个参数处指定

// ========
void clear(); // 删除所有元素
iterator erase(iterator pos); // 删除pos迭代器所指的元素，返回下一个元素的迭代器
iterator erase(beg, end); // 删除区间[beg, end)内的所有元素，返回下一个元素的迭代器
erase(keyElem); // 删除容器中key为keyElem的对组
```

查找操作:

```
iterator find(T1 key);
// 查找键key是否存在，若存在，返回该键的元素的迭代器；若不存在，返回map.end()
int count(T1 keyElem);
// 返回容器中key为keyElem的对组个数，对map来说只可能是0或1，对于multimap可能大于1

iterator lower_bound(T keyElem);
// 返回第一个key>=keyElem元素的迭代器
iterator upper_bound(T keyElem);
// 返回第一个key>keyElem元素的迭代器
pair<iterator, iterator> equal_range(T keyElem);
// 返回容器中key与keyElem上相等的两个上下限迭代器
```

容器小结

vector 可以涵盖其他所有容器的功能，只不过实现特殊功能时效率没有其他容器高。但如果只是简单存储，vector效率是最高的。
deque 相比于 vector 支持头端元素的快速增删。
list 支持频繁的不确定位置元素的移除插入。
set 会自动排序。
map 是元素为键值对组并按键排序的set。

vector 与 deque 的比较：
1.vector.at()比deque.at()的效率高；比如vector.at(0)是固定的，deque的开始位置是不固定的

2.如果有大量释放操作的话，vector花的时间更少

3.deque支持头部的快速插入与快速删除，这是deque的优点