语法笔记

STL

各种容器的成员函数

vector

使用条件

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

初始化

```
vector<int> v(10);  // 创建10个元素的数组v，初始化为默认值(0)
vector<int> v(10, 2);  // 创建10个元素的数组v，初始化为2
vector<int> v = {1, 2, 3, 4, 5};  // 使用数组初始化
vector<int> v2(v.begin(), v.end());
```

访问元素的方法

1. 通过下标：`v[i]`；
2. 通过迭代器：`*(v.begin() + i)`。

成员函数

```
vector& operator=(const vector& other);  // 赋值给容器
void assign(size_type count, const T& value);  // 可以对已经初始化的容器重新赋值
```

元素访问

```
reference at(size_type pos);  // 返回pos位置元素的引用，带边界检查
reference operator[](size_type pos);
reference front();  // 访问第一个元素
reference back();  // 访问最后一个元素
T* data();  // 返回指向内存中数组第一个元素的指针
```

迭代器

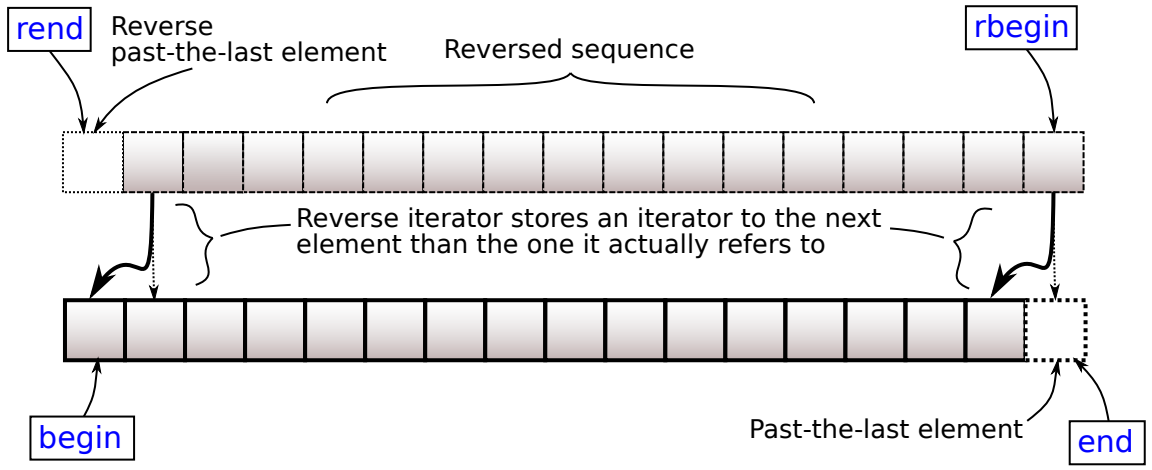
```
iterator begin();  // 返回指向起始的迭代器
iterator end();  // 返回指向末尾的迭代器
reverse_iterator rbegin();  // 返回指向起始的逆向迭代器
reverse_iterator rend();  // 返回指向末尾的逆向迭代器
```

容量

```
bool empty();  // 检查容器是否为空
size_type size();  // 返回容纳的元素数
size_type max_size();  // 返回根据系统或库实现限制的容器可保有的元素最大数量
void reserve(size_type new_cap);  // 增加vector的容量到大于或等于new_cap的值。不会修改size。若new_cap > capacity，则会重新分配存储空间，且所有迭代器失效。
size_type capacity();  // 返回容器当前已为之分配空间的元素数
```

修改器

```
void clear();  // 清除所有元素，size变为0，capacity不变。
iterator insert(iterator pos, const T& value);  // 插入元素，返回指向被插入value的迭代器
iterator insert(const_iterator pos, size_type count, const T& value);  // 插入count个value，返回第一个被插入元素的迭代器，若count == 0返回pos
iterator erase(iterator pos);  // 移除位于pos的元素
iterator erase(iterator first, iterator last);  // 移除范围[first, last)中的元素
void push_back(const T& value);  // 将元素添加到容器末尾，初始化新元素为value的副本
void push_back(T&& value);  // 移动value进新元素
template< class... Args >
constexpr reference emplace_back(Args&&... args);  // 将元素添加到容器末尾，返回被插入元素的引用
void pop_back();  // 移除容器的末元素
void resize(size_type count);  // 重设容器大小以容纳count个元素
void swap(vector& other);  // 将内容与other的交换
```

> 关于逆向迭代器的说明图和使用示例。
>
> 
>

```
    for (auto it = v.rbegin(); it != v.rend(); ++it) {  // 使用逆向迭代器遍历数组
        cout << *it << endl;
    }
```

stack

使用条件：

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

成员函数：

```
stack& operator=(const stack& other);  // 复制赋值运算符
stack& operator=(stack&& other);  // 移动赋值运算符
// 元素访问
reference top();  // 返回到栈顶元素的引用
// 容量
bool empty();
size_type size();  // 返回容纳的元素数
// 修改器
void push(const T& value);
void push(T&& value);
template< class... Args >
void emplace(Args&&... args);  // 在栈顶原位构造元素
void pop();
void swap(stack& other);  // 交换容器适配器与other的内容
```

queue

使用条件：

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

成员函数：

```
queue& operator=(const queue& other);
queue& operator=(queue&& other);
// 元素访问
reference front();
reference back();
// 容量
bool empty();
size_type size();
```

```
// 修改器
void push(const value_type& value);
void push(value_type&& value);
template< class... Args >
void emplace(Args&&... args);
void pop();
void swap(queue& other);
```

deque

使用条件:

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

成员函数:

```
deque& operator=(const deque& other);
deque& operator=(deque&& other);
void assign(size_type count, const T& value);
// 元素访问
reference at(size_type pos);  // 访问指定的元素，同时进行越界检查
regerence operator[](size_type pos);
reference front();
reference back();
// 迭代器
iterator begin();
iterator end();
reverse_iterator rbegin();
reverse_iterator rend();
// 容量
bool empty();
size_type size();
size_type max_size();
// 修改器
void clear();
iterator insert(iterator pos, const T& value);
iterator insert(iterator pos, size_type count, const T& value);
template< class... Args >
iterator emplace(const_iterator pos, Args&&... args);
iterator erase(iterator pos);
iterator erase(iterator first, iterator last);
void push_back(const T& value);
void push_back(T&& value);
template< class... Args >
reference emplace_back(Args&&... args);
void pop_back();
void push_front(...);
```

```
reference emplace_front(...);
void pop_front();
void resize();
void swap(deque& other);
```

priority_queue

使用条件：

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

成员函数：

```
priority_queue& operator=(const priority_queue& other);
priority_queue& operator=(priority_queue&& other);
// 元素访问
const_reference top() const;
// 容量
bool empty() const;
size_type size() const;
// 修改器
void push(const T& value);
template< class... Args >
void emplace(Args&&... args);
void pop();
void swap(priority_queue& other);
```

优先级的设置

默认提供最大元素的查找。
对于基本数据类型如`int`，`char`，`double`，可以直接使用。
区分`less`与`greater`：从堆根开始向下比较，大根堆值变小，小根堆值变大。堆排序中数组升序排列时要使用大根堆。

```
priority_queue<int> q;  // 大根堆
priority_queue<int, vector<int>, less<int>> q;  // 大根堆
priority_queue<int, vector<int>, greater<int>> q;  // 小根堆
```

对于结构体，通过重载`<`实现。例如：

```
struct fruit {
        string name;
        int price;
        friend bool operator<(const fruit& f1, const fruit& f2) {  // 不能省略const
                return f1.price < f2.price;  // 这里改为>则是价格低的水果优先级高
```

```
        }
        // 或
        friend bool operator<(fruit f1, fruit f2) {
        return f1.price < f2.price;
    }
        // 另一种方法
        bool operator<(const fruit& f) const {  // 两个const都不能省略
                return price < f.price;
        }
};
priority_queue<fruit> q;  // 价格高的水果优先级高
```

也可以单独使用结构体实现比较函数。

```
struct cmp {  // 效果等同于重载<
        bool operator()(fruit& f1, fruit& f2) {
                return f1.price < f2.price;  // 改成>则是价格低的优先级高
        }
};
priority_queue<fruit, vector<fruit>, cmp> q;  // 这样实现的cmp效果为价格高的优先级高
```

string

使用条件

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

初始化/构造函数：

```
string str = "Hello World.";  // 直接用=初始化
string(size_type count, char ch);  // 长度为count，用字符ch填充
string(const char* s, size_type count);  // 以s所指向的字符串的首count个字符构造string
string(const char* s);  // 以首个空字符('\0')确定字符串的长度
string(string& other, size_type pos, size_type count = string::npos);  // 以other的子串
[pos, pos+count)构造string，若不指定count，范围是[pos, other.size())
string(const string& other);  // 复制构造函数
```

常量：

```
static const size_type npos = -1;  // 对于无符号整数相当于最大值
```

成员函数：

```
string& operator=(const string& str);
string& operator=(const char* s);
string& operator=(char ch);  // 不能在初始化时使用
string& assign(size_type count, char ch);
string& assign(const string& str);
string& assign(const string& str, size_type pos, size_type count = npos);
string& assign(const char* s, size_type count);  // [s, s + count)
string& assign(const char* s);
```

元素访问

```
char& at(size_type pos);  // 访问指定字符，有边界检查
char& operator[](size_type pos);  // 访问指定字符
char& front();
char& back();
const char* data() const;  // 返回指向字符串首字符的指针
const char* c_str() const;  // 同上
```

迭代器

```
iterator begin();
iterator end();
reverse_iterator rbegin();
reverse_iterator rend();
```

容量

```
bool empty();
size_type size();
size_type length();
size_type max_size();  // 返回string由于保有系统或库实现限制所能保有的最大元素数
void reserve(size_type new_cap);
size_type capacity();
void shrink_to_fit();  // 请求移除未使用的容量。这是减少capacity()到size()的非强制请求，是否满足请求依赖于实现
```

操作

```
void clear();
string& insert(size_type index, size_type count, char ch);
string& insert(size_type index, const char* s);  // 以空字符'\0'结束
string& insert(size_type index, const char* s, size_type count);  // [s,s+count)可以含有空字符
string& insert(size_type index, const string& str);
string& insert(size_type index, const string& str, size_type index_str, size_type count);
```

```
iterator insert(iterator pos, char ch);
iterator insert(iterator pos, size_type count, char ch);
string& erase(size_type index = 0, size_type count = npos);  // 返回*this
iterator erase(iterator pos);  // 返回被擦除字符后随字符的迭代器或end()
iterator erase(iterator first, iterator last);  // 擦除[first, last)，返回擦除前last指向
字符的迭代器或end()
void push_back(char ch);
void pop_back();
string& append(size_type count, char ch);
string& append(const string& str);
string& append(const string& str, size_type pos, size_type count);
string& append(const char* s, size_type count);
string& append(const cahr* s);
string& operator+=(const string& str);
string& operator+=(char ch);
string& operator+=(const char* s);
bool starts_with(char ch);  // 检查ch是否为前缀
bool starts_with(const char* s);  // 检查空终止字符串s是否为前缀
bool ends_with(char ch);  // 检查后缀
bool ends_with(const char* s);
string substr(size_type pos = 0, size_type count = npos);  // 子串[pos, pos+count)
size_type copy(char* dest, size_type count, size_type pos = 0);  // 复制子串[pos,
pos+count)到dest所指向的字符串
void resize(size_type count);
void resize(size_type count, char ch);
void swap(string& other);  // 交换string的内容
// 以新字符串替换[pos, pos + count)或[first, last)所指示的string部分
string& replace(size_type pos, size_type count, const string& str);
string& replace(const_iterator first, const_iterator last, const string& str);
string& replace(size_type pos, size_type count, const basic_string& str,
                                size_type pos2, size_type count2 = npos);
string& replace(size_type pos, size_type count, const char* cstr, size_type count2);
string& replace(const_iterator first, const_iterator last,
                                const char* cstr, size_type count2);
string& replace(size_type pos, size_type count, const char* cstr);
string& replace(const_iterator first, const_iterator last, const char* cstr);
string& replace(size_type pos, size_type count, size_type count2, char ch);
string& replace(const_iterator first, const_iterator last, size_type count2, char ch);
// 比较2个字符序列
int compare(const string& str);
int compare(size_type pos1, size_type count1, const string& str);
int compare(size_type pos1, size_type count1, const basic_string& str,
                        size_type pos2, size_type count2 = npos);
int compare(const char* s);
int compare(size_type pos1, size_type count1, const char* s);
int compare(size_type pos1, size_type count1, const char* s, size_type count2);
```

> `compare`：按下列方式比较始于 `data1` 的 `count1` 个字符组成的字符序列与始于 `data2` 的 `count2` 个字符组成的字符序列。首先，用 `size_type rlen = min(count1, count2)` 计算要

> 比较的字符数。然后调用 `Traits::compare(data1, data2, rlen)` 比较序列。对于标准字符特性，此函数进行逐字符字典序比较。若结果为零（到此为止的字符序列相等），则按下列方式比较其大小：
>
> | 条件 | | 结果 | 返回值 |
> | --- | --- | --- | --- |
> | `Traits::compare(`*data1*`, `*data2*`, `*rlen*`) < 0` | | *data1* 小于 *data2* | <0 |
> | `Traits::compare(`*data1*`, `*data2*`, `*rlen*`) == 0` | *size1* < *size2* | *data1* 小于 *data2* | <0 |
> | | *size1* == *size2* | *data1* 等于 *data2* | 0 |
> | | *size1* > *size2* | *data1* 大于 *data2* | >0 |
> | `Traits::compare(`*data1*`, `*data2*`, `*rlen*`) > 0` | | *data1* 大于 *data2* | >0 |

set, unordered_set

map, unordered_map

pair

bitset

头文件algorithm

标准输入输出

cin
getline
getchar
putchar
gets
puts

常用头文件

cstdlib

cctype

cmath

complex

其他

数与字符串的转换

字符串转为数

```
#include <cstdlib>

double atof(const char* str);
int atoi(const char* str);
long int atol(const char* str);
long long int atoll(const char* str);
```

`string`库的函数，调用`cstdlib`中的`std::strtol`，`std::strtoll`等。

```
#include <string>
int stoi(const std::string& str, std::size_t* pos = 0, int base = 10);
long stol(const std::string& str, std::size_t* pos = 0, int base = 10);
long long stoll(const std::string& str, std::size_t* pos = 0, int base = 10);
unsigned long stoul(const std::string& str, std::size_t* pos = 0, int base = 10);
unsigned long long stoull(const std::string& str, std::size_t* pos = 0, int base = 10);
```

> 舍弃所有空白符（以调用`std::isspace()`鉴别），直到找到首个非空白符，然后取尽可能多的字符组成底base的无符号整数表示，并将它们转换成一个整数值。合法的无符号整数值由下列部分组成：
>
> 1. (可选)正或负号
> 2. (可选)指示八进制底的前缀（`0`）（仅当底为 8 或 0 时应用）
> 3. (可选)指示十六进制底的前缀（`0x`或`0X`）（仅当底为 16 或 0 时应用）
> 4. 一个数字序列
>
>   底的合法集是 $\{0,2,3,\ldots,36\}$。合法数字集对于底 2 整数是`{0,1}`，对于底3整数是`{0,1,2}`，以此类推。对于大于`10`的底，合法数字包含字母字符，从对于底 11 整数的`Aa`到对于底36整数的`Zz`。忽略字符大小写。
>
>   若 base 为 0，则自动检测数值进制：若前缀为`0`，则底为八进制，若前缀为`0x`或`0X`，则底为十六进制，否则底为十进制。

```
float stof(const std::string& str, std::size_t* pos = 0);
double stod(const std::string& str, std::size_t* pos = 0);
long double stold(const std::string& str, std::size_t* pos = 0);
```

> 函数会舍弃任何空白符（由`std::isspace()`确定），直至找到首个非空白符。然后它会取用尽可能多的字符，以构成合法的浮点数表示，并将它们转换成浮点值。合法的浮点值可以为下列之一：
> **十进制**浮点数表达式。它由下列部分组成：
>
> 1. (可选) 正或负号
> 2. 非空的十进制数字序列，可选地包含一个小数点字符（定义有效数字）

3. (可选) `e` 或 `E` ，并跟随可选的正或负号，以及非空十进制数字序列（以 10 为底定义指数）
   二进制浮点数表达式。它由下列部分组成：
4. (可选) 正或负号
5. `0x` 或 `0X`
6. 非空的十六进制数字序列，选地包含一个小数点字符（定义有效数字）
7. (可选) `p` 或 `P` ，并跟随可选的正或负号，以及非空十进制数字序列（以 2 为底定义指数）
   无穷大表达式。它由下列部分组成：
8. (可选) 正或负号
9. `INF` 或 `INFINITY` ，忽略大小写
   非数（NaN）表达式。它由下列部分组成：
10. (可选) 正或负号
11. `NAN` ，忽略 `NAN` 部分的大小写。

格式化读取字符串内容的例子：

```
#include <cstdio>

char info[] = "ID: 120 Name: John Age: 25";
int ID;
char name[20];
int age;
sscanf(info, "ID: %d Name: %s Age: %d", &sno, name, &age);
```

数转为字符串

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

string to_string(int n);
string to_string(int val);
string to_string(long val);
string to_string(long long val);
string to_string(unsigned val);
string to_string(unsigned long val);
string to_string(unsigned long long val);
string to_string(float val);
string to_string(double val);
string to_string(long double val);
```

格式化输出到字符串的例子：

```
#include <cstdio>

char buffer[50];
int num = 10;
sprintf(buffer, "Value of num = %d.", num);
```

格式说明符：

- **%d**: 读取整数。
- **%f**: 读取浮点数。
- **%s**: 读取字符串。
- **%c**: 读取字符。
- **%x**: 读取十六进制整数。
- **%o**: 读取八进制整数。
- **%u**: 读取无符号整数。