

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

```

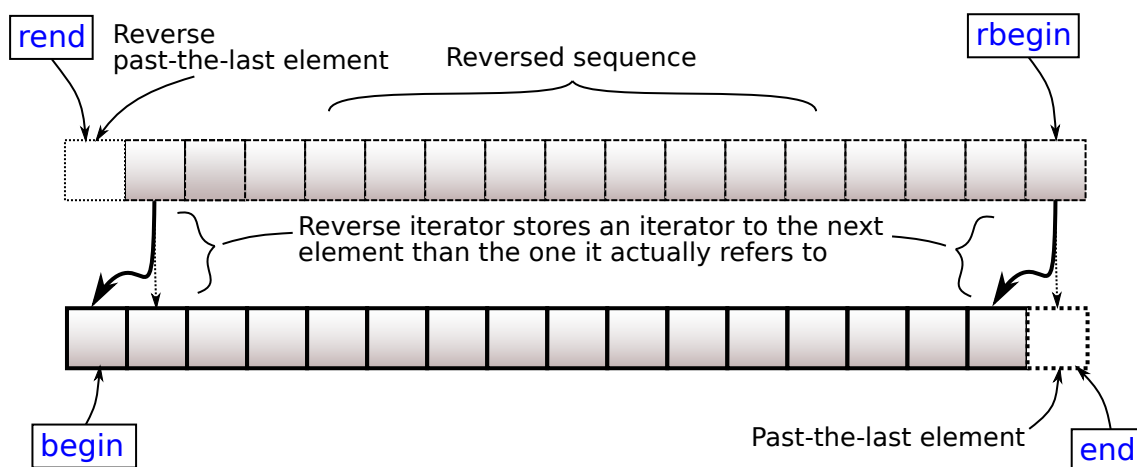
修改器

```

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
reference 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 char* 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
<code>Traits::compare(data1, data2, rlen) < 0</code>		<i>data1</i> 小于 <i>data2</i>	<0
<code>Traits::compare(data1, data2, rlen) == 0</code>	<i>size1</i> < <i>size2</i>	<i>data1</i> 小于 <i>data2</i>	<0
	<i>size1</i> == <i>size2</i>	<i>data1</i> 等于 <i>data2</i>	0
	<i>size1</i> > <i>size2</i>	<i>data1</i> 大于 <i>data2</i>	>0
<code>Traits::compare(data1, data2, rlen) > 0</code>		<i>data1</i> 大于 <i>data2</i>	>0

set, unordered_set

set

使用条件：

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

成员函数：

```
set& operator=(const set& other);
// 迭代器
iterator begin();
iterator end();
reverse_iterator rbegin();
reverse_iterator rend();
// 容量
bool empty();
size_type size();
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
// 修改器
void clear();
std::pair<iterator, bool> insert(const value_type& value);
template< class... Args >
std::pair<iterator, bool> emplace(Args&&... args);
iterator erase(iterator pos); // 返回后随最后被移除的元素的迭代器
iterator erase(iterator first, iterator last); // 移除范围[first, last)中的元素
size_type erase (const key_type& key); // 删除值为key的元素，返回成功删除的个数
void swap(set& other);
// 查找
size_type count(const Key& key); // 返回匹配特定键的元素数量，因为容器不允许重复故为1或0
iterator_find(const Key& key);
bool contains(const Key& key); // 检查容器是否含有带特定键的元素
iterator lower_bound(const Key& key); // 返回指向首个不小于key的元素的迭代器，若找不到返
```

```

return end();
iterator upper_bound(const Key& key); // 返回指向首个大于key的元素的迭代器，若找不到返回
end();

```

unordered_set

使用条件：

```

#include <unordered_set>
using namespace std;

```

成员函数：

```

unordered_set& operator=(const unordered_set& other);
// 迭代器
const_iterator begin() const;
const_iterator end() const;
// 容量
bool empty();
size_type size();
size_type max_size();
// 修改器
void clear();
std::pair<iterator, bool> insert(const value_type& value);
template< class... Args >
std::pair<iterator, bool> emplace(Args&&... args);
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);
size_type erase(const key_type& key);
void swap(unordered_set& other);
// 查找
size_type count(const Key& key); // 返回匹配特定键的元素数量，因为容器不允许重复故为1或0
iterator find(const Key& key);
bool contains(const Key& key);

```

map, unordered_map

map

使用条件：

```

#include <map>
using namespace std;

```

元素访问：

```
class Key; // 键
class T; // 值
map<Key, T> mp; // 定义容器
```

1. 通过下标访问: `T& operator[](const Key& key)`
2. 通过at函数, 带边界检查: `T& at(const Key& key)`
3. 通过迭代器访问: `map<Key, T>::iterator it`, 通过 `it->first` 访问键, `it->second` 访问值。
例子:

```
map<char, int> mp;
mp['m'] = 1; // 可以直接插入元素
mp['p'] = 2;
for (map<char, int>::iterator it = mp.begin(); it != mp.end(); ++it) {
    cout << it->first << " : " << it->second << endl;
}
```

成员函数:

```
map& operator=(const map& other);
T& operator[](const Key& key); // 返回到映射到等于key的键的值的引用, 若键不存在则进行插入
// 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 erase(iterator pos);
iterator erase(iterator first, iterator last);
size_type erase(const key_type& key);
void swap(map& other)
// 查找
size_type count(const Key& key); // 返回匹配特定键的元素数量, 容器不允许重复故为1或0
iterator find(const Key& key);
bool contains(const Key& key);
std::pair<iterator, iterator> equal_range(const Key& key); // 返回容器中所有拥有给定键的元素范围。范围以2个迭代器定义, 一个指向首个不小于key的元素, 另一个指向首个大于key的元素
iterator lower_bound(const Key& key); // 返回指向首个不小于key的元素的迭代器
iterator upper_bound(const Key& key); // 返回指向首个大于key的元素的迭代器
```

unordered_map

使用条件：

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

成员函数：

```
unordered_map& operator=(const unordered_map& other);
// 迭代器
iterator begin();
iterator end();
// 容量
bool empty();
size_type size();
size_type max_size();
// 修改器
void clear();
iterator erase(const_iterator pos);
iterator erase(const_iterator first, const_iterator last);
size_type erase(const key_type& key);
void swap(unordered_map& other);
// 查找
T& at(const Key& key);
T& operator[](const Key& key);
size_type count(const Key& key); // 返回1或0
iterator find(const Key& key);
bool contains(const Key& key);
std::pair<iterator, iterator> equal_range(const Key& key); // 返回容器中所有键等于key的元素范围。范围以2个迭代器定义，第一个指向所需范围的首元素，而第二个指向范围的尾后一位元素
```

pair

`std::pair` 是类模板，提供在一个单元存储两个相异类型对象的途径，是 `std::tuple` 的拥有两个元素的特殊情况。

使用条件：

```
#include <utility> // 在map头文件中会被引用
using namespace std;
```

成员对象：

```
template<class T1, class T2>
struct pair {
    T1 first;
```

```
T2 second;

};

pair<int, int> p;
```

直接使用 `p.first` 和 `p.second` 访问元素。

成员函数：

```
pair();
pair(const T1& x, const T2& y);
template< class U1, class U2 >
pair(const pair<U1, U2>& p);
pair& operator=(const pair& other);
template< class U1, class U2 >
pair& operator=(const pair<U1,U2>& other);
void swap(pair& other);
```

非成员函数：

```
// 构造std::pair对象，从参数类型推导目标类型
template< class T1, class T2 >
constexpr std::pair<V1,V2> make_pair(T1&& t, T2&& u);
```

应用：

1. 替代二元结构体及其构造函数；
2. 作为map的键值对插入。

```
map<string, int> mp;
mp.insert({ "alpha", 1 });
mp.insert(make_pair("beta", 2));
mp.insert(pair<string, int>("gamma", 3));
```

bitset

类模板 `bitset` 表示一个 `N` 位的固定大小序列。可以用标准逻辑运算符操作位集，并将它与字符串和整数相互转换。模板形参 `N` 是要为 `bitset` 分配存储的位数。

使用条件：

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

bitset<100> bset; // 分配100位
```

成员函数：

```

bitset(); // 默认构造函数。构造所有位设为零的 bitset
bitset(unsigned long long val); // 构造bitset, 初始化其首(最右、最低)M位位置为对应val
的位值, 其中M是unsigned long long的位数和正在构造的bitset中的位数N的较小者。其他位赋0
bitset(const char* str, typename std::string::size_type n = std::string::npos,
        char zero = CharT('0'), char one = CharT('1')); // 使用字符串构造, n为使用字符数
bool operator==(const bitset<N>& rhs) const; // 若*this与rhs中的所有位相等则返回true
// 元素访问
bool operator[](std::size_t pos) const;
reference operator[](std::size_t pos);
bool test(size_t pos) const; // 返回位于位置pos的位的值
bool all(); // 检查是否全部位被设为 true
bool any(); // 检查是否任一位被设为 true
bool none(); // 检查是否无位被设为 true
std::size_t count() const; // 返回被设为true的位数
// 容量
std::size_t size() const; // 返回bitset所能保有的位数, 即模板形参N
// 修改器
bitset<N>& operator&=(const bitset<N>& other); // 二进制与
bitset<N>& operator|=(const bitset<N>& other); // 二进制或
bitset<N>& operator^=(const bitset<N>& other); // 二进制异或
bitset<N> operator~() const; // 二进制非
bitset<N> operator<<(std::size_t pos) const; // 二进制左移
bitset<N>& operator<<=(std::size_t pos); // 当前对象左移
bitset<N> operator>>(std::size_t pos) const; // 二进制右移
bitset<N>& operator>>=(std::size_t pos); // 当前对象右移
bitset<N>& set(); // 设置所有位为true
bitset<N>& set(std::size_t pos, bool value = true); // 设置pos位为value
bitset<N>& reset(); // 设置所有位为false
bitset<N>& reset(std::size_t pos); // 设置pos位为false
bitset<N>& flip(); // 翻转所有位
bitset<N>& flip(std::size_t pos); // 翻转pos位
// 转换
std::string to_string(char zero = '0', char one = '1') const; // 转换bitset的内容为
string。用zero表示拥有值false的位, 以one表示拥有值 true 的位。产生的字符串含N个字符, 其首
字符对应最末(第N-1)位, 其尾字符对应首位。
unsigned long to_ulong() const; // 转换bitset的内容为unsigned long整数。bitset的首位对应
数的最低位, 而尾位对应最高位。
unsigned long long to_ullong() const; // 转换bitset的内容为unsigned long long整数。
bitset的首位对应数的最低位, 而尾位对应最高位。

```

注意:

1. `&=`、`|=`、和 `^=` 仅对拥有相同大小 `N` 的 `bitset` 定义。
2. 返回值是否带引用, 带引用可能会修改原位集。

非成员函数:

```

template< std::size_t N >
bitset<N> operator&(const bitset<N>& lhs, const bitset<N>& rhs); // 二进制与

```

```
template< std::size_t N >
bitset<N> operator|(const bitset<N>& lhs, const bitset<N>& rhs); // 二进制或
template< std::size_t N >
bitset<N> operator^(const bitset<N>& lhs, const bitset<N>& rhs); // 二进制异或
```

头文件algorithm

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 $\{0, 2, 3, \dots, 36\}$ 。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 <stdio>

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 <stdio>

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。