# STL

## STL

### vector

支持对元素的下标访问，在尾部添加和删除元素效率高

insert() erase() push\_back()

front( ) back() pop\_back()

begin() end()

======与数组转换

int a[5]={1,2,3,4,5}; v(a,a+5);

//a代表begin迭代器 a+5代表end迭代器（最后一个元素后面）

int \*p = &a[0];

========赋值

vector<int> v;//定义一个空的vector对象

vector<int> v(5); //设置为5个0，相当于调用int的默认构造

vector<int> v(5,10); 5个10

v.resize(5); //调整vector大小，如果原来有内容则保留原来内容，如果没有，则调用默认构造创建 （只对新分配的内存进行赋值）

v.resize(5,10);//调整大小，并对新分配的内容设置值

v.assign(5,3);//清空原有内容，并赋值为3

list<int> l(3,5);

v.assign(l.begin(),l.end());//将list赋值给vector

=======操作

v.reserve(100);//设置预留空间大小，与capacity相关

vector<int>::size\_type size = v.capacity();

遍历取值

for(int i=0;i<v.size();i++) { cout<<v[i]<<endl; }

元素存取

v.push\_back(x);

v.pop\_back() ;

int end=v.back();

int begin = v.front();

try{x = v.at(100);}catch(...){}//比使用v[100]安全，可以抛出异常而不让程序崩溃

v.clear();只清空内容，不释放分配的内存，内存到结束时释放，下标索引数量变为0。因为内存内容还在，release正常，debug会崩溃。

v.swap(vector<int>());清空内容，释放内存

===========迭代器

任何可能导致容器结构发生变化的函数被调用后，先前获得的迭代器可能失效，需要重新初始化

erase和insert操作单个元素时，都返回操作位置的迭代器。

删除[iterA – iterB)时，返回iterB，vector不可使用范围删除的迭代器返回值。

reverse\_iterator rbegin() rend()

const\_iterator

const\_reverse\_iterator

vector<int>::iterator it;

定义了一个vector中的迭代器it，迭代器是模版的共有内置类

通过迭代器遍历容器中元素：

for(it=v.begin();it!=v.end();it++)

cout<<\*it<<endl;

位置指针要用it迭代器

v.insert(位置指针，数量（可省略），数值)；

//在指针位置前插入，插入效率比list低

v.erase(位置指针) erase(开始位置，结束位置)

### list

在任何位置插入和删除都很方便，不支持元素下标访问

insert() erase()

push\_front() push\_back()

front() back() pop\_back()

pop\_front() begin() end()

除了不支持vector的下标操作外，和vector的初始化、插入、删除等函数的使用相同。

list.sort(); //可以让list中的元素从小到大自动排序，如果是元素是类类型，需要重载＜号

//删除满足某个条件的元素

bool equal(int &temp){return temp == 3;}

l.remove\_if(equal);

l.remove\_if([](int &temp)->bool{return temp == 3;});

### map

以key的升序（插入时排序）存储key\_value对的序列，每个key只能出现一次，根据key提取value的效率最高，排序根据key类型的‘<’操作符对其进行比较

map<string,int> m;

m["a"]=1; //m[\*\*]可以访问到key对应的成员（[]操作符返回值相当于value），同时也是进行检查然后插入key的操作。

m.insert(pair<string,int>("a",1));

m.insert(map<string,int>::value\_type("b",2));

map<int,int> m;

pair<map<int,int>::const\_iterator,bool> result= m.insert(map<int,int>::value\_type(1,2));//insert的返回值第二个表示是否插入成功，第一个参数指向准备插入的内容。

如果要连续调用map，可以使用引用，减少插入key的检查操作：

struct A{int x, int y};

map<int , map<int ,A>> m;

map<int , map<int ,A>> &mx = m[1][0];

mx.x = 1;

mx.y = 2;

map<string,int>::iterator it;

===========遍历

for(it=m.begin();it!=m.end();it++)

{ cout<<it->first<<endl<<it->second<<endl; }

===========查找

it=m.find("a");

if(it!=m.end())

{ cout<<m["a"]<<endl; }

iterator erase(iterator it); //通过一个条目对象删除

iterator erase(iterator first, iterator last); //删除一个范围

size\_type erase(const Key& key); //通过关键字删除，返回的是删除的个数，只会是0或者1 。multimap值可以是大于1。

========== multimap

multimap<int,string> m;

int key = 1;

m.insert(pair<int,string>(key,"a"));

m.insert(pair<int,string>(key,"aa"));//相同键可以对应多个值

m.insert(pair<int,string>(2,"b"));

//查找并遍历某个键的值（通过count统计某个键对应的值的个数）

multimap<int,string>::const\_iterator iter = m.find(key);

for(size\_t count = 0; count != m.count(key); ++count,++iter)

{ cout<<iter->second<<endl; }

C11初始化：

map<int, int> mm = {{1,2},{3,4} };

### bitset

#include <bitset>

using namespace std;

左边存储序列低位，右边存储高位

bitset<10> b(8);

//能存储10个位数据，用8进行初始化。也可以用b=8

//存储内容为0001000000

for(int i=b.size()-1 ; i>=0 ; i--)

{ cout<<b[i]<<endl; }

string s("1100");

bitset<10> b(s); //也可以使用string串形式的二进制数字初始化

可以进行位操作： bitset<10> b(5); bitset<10> a(2); b=a|b;

any() 判断是否存在1

none() 是否全0

count() 1的个数

os<<b 输入到os流，流内会正向排序

flip() 逐位取反，flip(pos)只对某位进行取反

test(pos) 测试pos位置是否为1

set() 全部置1，可用set(pos)

reset() 全部置0，可用reset(pos)

### 其他容器

deque和stack可以做适配器，stack可以用deque初始化

deque<int> d(v.begin(),v.end());

stack<int> s(d);

==================双端队列(deque)

支持对元素下标访问，支持在两端添加或者删除元素

insert() erase()

push\_front() push\_back()

front() back()

pop\_front() pop\_back()

begin() end()

==================堆栈（stack）

只支持在一段存储和提取元素，无iterator

push()

top()

pop()

==================队列（queue）

支持从前端提取，从后端插入元素

pop() front() back() push()

==================优先队列（priority\_queue）

类似于队列，但所提取的是具有最高优先级的元素

pop() top() push()

==================集合（set）

映射的低级形式，不需为每个key指定一个value

s.insert(1); //直接放入元素

\*it 可以直接对迭代器取值

==================多重映射（multimap）

映射的通用形式，一个key可以拥有超过一个的key\_value对

==================多重集合（multiset）

多重映射的低级形式，不需要为每个key指定一个value

## 循环队列

#define QueSize 11

class MyQue

{

public:

MyQue()

{

dat.resize(QueSize,0);

front = back = 0;

}

bool push(int\* element)

{//向队列插入元素

if ((back + 1) % QueSize == front) //判断队列是否满

{

return false;

}

else

{

dat[back] = \*element;

back = (back+1) % QueSize;

return true;

}

}

int\* pop()

{//从队顶弹出元素

int \*pRetElement = NULL;

if (front != back)//判断队列是否为空

{

pRetElement = &dat[front];

front = (front + 1) % QueSize;

}

return pRetElement;

}

public:

int AvailableLength()

{//返回队列的可以用空间长度

return QueSize - 1 - (back - front + QueSize) % QueSize ;

}

int UsedLength()

{//返回队列的已使用空间长度

return (back - front + QueSize) % QueSize;

}

private:

int front; //对头指针

int back; //队尾指针

vector<int> dat;

};

## <algorithm>

#### 排序

如果stl的成员变量是类类型，需要重载类的<号

//假设要比较A内的int x;成员

bool operator<( const A& other) const

{

if(x<other.x)

return true; //代表交换两个元素

else

return false;

}

对数据排序:std::sort(v.begin(),v.end());

插入时排序：v.insert(std::upper\_bound(v.begin(),v.end(),a1),a1);

#### 距离

size\_t i = std::distance(v.begin(),v.end());

可以返回两个迭代器之间的距离（此处返回了v的长度）

#### 去重

std::sort(v.begin(),v.end());

//将数据排成1,2,2,3,3,4,5

vector<int>::iterator iter = std::unique(v.begin(),v.end());

//将不重复的元素向前赋值，数据变成1,2,3,4,5,4,5，并返回4的位置（返回不重复位置的末尾）

v.erase(iter,v.end());

或者：std::sort(v.begin(),v.end());

v.resize(std::distance(v.begin(),std::unique(v.begin(),v.end())));

#### 查找

vector<int>::iterator iter = std::find(v.begin(),v.end(),10);

查找到则iter执行找到的元素，查找不到则指向v.end();

如果是类类型，则需要重载==符号。

#### 智能指针

std::auto\_ptr<A> p(new A());

p->fun(); //使用指针

指针A在不需要使用时，会自动释放

p.reset(); //手动释放指针内存

p.reset(pb);//释放之前的指针，同时让p接管另一个指针

if(0 != p.get() ) {p->fun();}

对智能指针进行非空判断

## lambda表达式

lambda表达式的参数列表基本和函数的一致，不过有如下限制：

参数列表不能有默认参数

不能是可变参数列表

所有的参数必须有个变量名

遍历每个元素，让该元素\*10；

struct A{ void operator()(int &x){x = x\*10;} }; //需要重载()

std::for\_each(v.begin(),v.end(),A()); // vector<int> v;

//使用lambda

for\_each(v.begin(), v.end(), [](int &a) { a \*= 10;});

struct Data

{

Data():a(1),b(1){}

int a;

int b;

};

vector<Data> v(5);

for\_each(v.begin(), v.end(), [](Data &dat) { dat.a \*= 10;});//访问每个元素的引用

for\_each(v.begin(), v.end(), [](Data dat) {cout << dat.a << endl;});//访问每个元素的值

//////////访问外部变量

int sum = 0;

for\_each(v.begin(), v.end(), [&](Data &dat) { sum += dat.a ;});//按引用访问外部元素

int temp = 100;

for\_each(v.begin(), v.end(), [=](Data dat) {cout << dat.a \* 100 << endl;});//按值访问外部元素

//作为参数传递

template<typename \_Fun>

void fun(\_Fun callback)

{

int x = 10;

callback(x);

}

调用：

int factor = 10;

fun([&] (int &x){

cout << factor\* x << endl;

});

//map访问方式

map<int, Data> m;

m[1] = Data();

m[2] = Data();

for\_each(m.begin(), m.end(), [](pair<int, Data> dat) { cout << dat.first << " " << dat.second.a << endl;});

//返回值

int z = []()->int { return 10; }();

()->int可以省略

//设置数组的值

const int SIZE = 20;

int array[SIZE];

generate\_n(array, SIZE,[]() { return rand() % 10; });

## <functional>

int fun1(int a,double b){return a + b;}

auto fun2 = [](int a, double b) {return a - b; };

class Functor

{

public:

int operator() (int a,double b){return a\*b;}

};

class ABC

{

int x;

public:

ABC() { x = 1000; }

int fun3(int a, double b) { return a/b + x; }

};

void \_\_fun(int a, std::function<int(int, double)> fun)

{

double dd = 100.2;

int ret = fun(a, dd);

cout << ret << endl;

}

\_\_fun(1, fun1);

\_\_fun(1, fun2);

int x = 5;

\_\_fun(1, [&](int a, double b) {return x + a + b; });

\_\_fun(1, Functor());

ABC aa;

std::function<int(int,double)> obj = std::bind(&ABC::fun3, &aa, std::placeholders::\_1,std::placeholders::\_2);

\_\_fun(1, obj);

## <memory>

auto\_ptr<A> pa(new A);

A \*p = pa.get();

不能传入数组，数组要用delete[]

## 模板

template <typename T>

此行下的是模版，且有未实例化的T，<>中间的是类型形参表

当类型形参表多个时：<typename T, typename F>

模版不能像普通函数那样在.h中只声明，.cpp文件中实现。因为不是一个具体类型，所以只声明会出错，因此只能将声明和实现都放在头文件中

typename Element::ID\_TYPE id;

可以直接使用另一个模版类中的类型。

### 函数模版

template <typename T>

void ab(T a, T b){1}

函数必须要传入T才可以使用（不可以在内部定义T）

1. ab(100,200) 默认隐式推断（类模版不能用此法）
2. ab<int>(100,200)

特化：（相当于重载，函数名相同）

template<> 代表特化后的模版内无剩余的未实例化的参数

void ab(int a, int b){2} 用int特化

### 类模版

template<typename T>

class A{

T x;

void ab(){1}

};

使用时： A<int> a;

template<>

class A<int>{

int x;

void ab(){2}

};

类模版不是类，因为有未定义的typename T，要实例化（编译时）T才是类。编译时，无栈参与，栈在运行时参与，变量存于栈内，常量存代码段

### 常量模板

非类型实参只能是常量性质

template<int a，int b>

class A

{

void fun(){int x = a; x += b ; cout<<x<<endl;}

};

可以在类内使用未知的常量

### 模板的派生

基模版派生子模版：

template <typename T>

class A{T t;..........};

template <typename T>

class B:public A<T>{........}；

子模版继承父类：

子类：template <typename T，typename Base>

class B:public Base{....}；

父类：class A{....};

使用时： B<int ,A> b;

父类如果是模版类时：

子类改为：

template<typename T, template<typename F>Class Base,

typename BA>

class B:public Base<BA>{........};

用BA使Base变为类

模版中定义模版函数：

template <typename T>

class A{public:

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

template<typename T1>

void ab(T1 b){.........}

};

使用时：

A<int> ().ab<int>(3);

创建一个临时对象，在调用ab模版函数（用int特化）

可用 A<int> a;

此时其中模版函数不可用

类外定义时：

template <typename T>

template<typename T1> 模版下的模版

void A<T>::ab(T1 b){.....}

类中类：

A<int> a;

A<int>::B<long> b;

### 特化

当类型参数表有多个时，进行多种特化后，在使用时选择特化程度最高的

针对指针的特化：

template <typename T>

class A<T\*>{}

template <typename T>

class A<T[]>{}

### 外部实现

可以内部定义，外部实现（同一个.h文件中）

template <typename T>

void A<T>::ab(T t){}

类的成员特化：

template<>

void A<char>::ab(T t){}

### 模版默认值

typename T=int 可以使用默认值

### 模版中获取类型

template<typename T>

class A

{

typename T::TY b;

};

class X

{

int a;

public:

typedef int TY;

};