泛型算法

- accumulate

```
vector<int> vect={1,2,3,4};
//操作数字
int sum = accumulate(vect.begin(),vect.end(),0);
cout<<sum<<endl;
//操作字符串
vector<string> _svect={"1,2,3,4","fjm","jfhiwoao"};
string s = accumulate(_svect.begin(),_svect.end(),string(""));
cout<<s<<endl;
```

- fill
- 将容器每个元素置0

```
fill(vect.begin(),vect.end(),0);
```

- back_inserter
- 创建一个插入迭代器
- ```
  vector<int> vect;
  //对vector赋值10个元素，值为10
  fill_n(back_inserter(vect),10,10);
  ```

- copy
- ```
  int a1[] = {1,2,3,4};
  int a2[4];
  //ret指向拷贝到cvect的尾元素之后的位置
  auto ret = copy(begin(a1),end(a1),a2);
  ```

- replace
- ```
  //将a1中所有值为2的元素替换为555
  replace(begin(a1),end(a1),2,555);
  ```

- sort
- ```
  //从大到小排序 降序
  sort(begin(a1),end(a1),greater<int>());
  //从小到大排序 升序
  sort(begin(a1),end(a1),less<int>());
  //升序
  sort(vect.begin(),vect.end());
  //降序
  sort(vect.crbegin(),vect.crend());
  ```

```
vector<string> vect = {"fbg","igfiuw","jnd","bfuyjsa"};
//按照长度从小到大排序
sort(vect.begin(),vect.end(),compare);
//按照字典序排序
stable_sort(vect.begin(),vect.end());
//先按长度排序，长度相同按字典序排序
stable_sort(vect.begin(),vect.end(),compare);
```

lamba表达式

- 默认情况下，从lamba生成的类都包含一个对应该lamba所捕获的变量的数据成员，lamba数据成员也在lamba对象创建时被初始化。

```
[capture list](parameter list) -> return type{function body}

auto f = []{return 42};

[](const string &a,const string &b){
    return a.size()<b.size();
}
//for_each循环
for_each(vect.begin(),vect.end(),[](const string &s){cout<<s<<"---";});
```

- 值捕获

```
int v = 42;
auto f = [&v]{return v;};
v = 0;
cout<<f()<<endl;
```

- 引用捕获

  当以引用方式捕获一个变量时，必须保证在lamba执行时变量是存在的。

```
int v = 42;
auto f = [&v]{return v;};
v = 0;
cout<<f()<<endl; //输出0
```

- upper_bound

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    //返回的是一行
    auto row = upper_bound(matrix.begin(),matrix.end(),
                           target,[](const int b,const vector<int> &a){
                               //b = target,就是传进来的target
                               return  b<a[0];
                           });
    if(row == matrix.begin()){
        return false;
    }
    --row;
    //行二分
    return binary_search(row->begin(),row->end(),target);
```

```
}
```

迭代器

- 只有容器支持push_front的情况下，我们才可以使用front_inserter。类似的，只有容器支持push_back的情况下，我们才能使用back_inserter。

1. back_inserter创建一个使用pu　sh_back的迭代器
2. front_inserter创建一个使用push_front的迭代器
3. inserter创建一个使用insert的迭代器。此函数必须接受第二个参数，这个参数必须是一个指向给定容器的迭代器。元素将被插入 到给定迭代器所表示的元素之前。

- 流迭代器
- 流迭代器不支持递减运算，因为不可能在一个流钟中反向移动
- istream_iterator

```
    istream_iterator<int> in_iter(cin);
    istream_iterator<int> in_eof; //默认初始化迭代器作尾后值使用的迭代器
    vector<int> vect;
    //输出流迭代器以分隔符 “--” 连接
    ostream_iterator<int> out_iter(cout,"--");
    while(in_iter != in_eof){
        auto e = *in_iter++;
        *out_iter++ = e;
        vect.push_back(e);
    }
    cout<<endl; //输出流
```

- 迭代器的类别

输入迭代器	只读，不写；单遍扫描，只能递增
输出迭代器	只写，不读；单遍扫描，只能递增
前向迭代器	可读写；多遍扫描，只能递增
双向迭代器	可读写；多遍扫描，可递增递减
随机访问迭代器	可读写，多遍扫描，支持及全部迭代器运算

- list和forward_list成员函数版本的算法 返回类型都为void

lst.merge(lst2) lst.merge(lst2, cmp)	**将来自lst2的元素合并入lst。lst和lst2都必须是有序的。元素将从ls2中删除。在合并之后，lst2变为空。**
lst.remove(val) lst.remove_if(pred)	调用erase删除掉与给定值相等
lst.reverse()	反转lst中元素的顺序
lst.sort() lst.sort(comp)	使用<或给定比较操作排序元素
lst.unique() lst.unique(pred)	调用erase删除同一个值得连续拷贝。

lst.splice(args)	//在list2中加入list1
	list2.splice(++list2.begin(),list1);

关联容器

map（按关键字有序保存元素）	**关联数组；保存关键字-值只对**
set（按关键字有序保存元素）	关键字即值，即保存关键字的容器
multimap（按关键字有序保存元素）	关键字可以重复的map
multiset（按关键字有序保存元素）	关键字可以重复出现的set
unordered_map(无序集合)	用哈希函数组织的map
unordered_set(无序集合)	用哈希函数组织的set
unordered_multimap(无序集合)	哈希组织的map;关键字可以重复出现
unordered_multiset(无序集合)	哈希组织的set;关键字可以重复出现

- pair

pair<T1,T2> p;	**p是一个pair，两个类型分别为T1和T2的成员都进行了值初始化**
pair<T1,T2> p(V1,V2);	p是一个成员类型为T1和T2；first和second成员分别用v1和v2
pair<T1,T2> p= {V1,V2};	等价p(V1,V2)
make_pair(v1,v2)	返回一个用v1和v2初始化的pair。pair的类型从v1和v2的类型推断出来
p.first	返回p的名为first的(共有)数据成员
p.second	返回p的名为second的(共有)数据成员
p1 relop p2	关系运算符（<,>,<=.>=）按字典序定义
p1 == p2	当first和second成员分别相等时，两个pair相等。
p1 != p2	判断利用元素的==运算符实现

- 关联容器额外的类型别名

key_type	**此容器类型的关键字类型**
mapped_type	每个关键字关联的类型：只适用于map
value_type	对于set，于key_type相同 对于map,为pair<const key_type,mapped_type>

- 当使用一个迭代器遍历一个map、multimap、set或multiset时，迭代器按关键字升序遍历元素
- 关联容器得insert的操作

  c.insert(v) //v是value_type类型的对象；args用来构造一个元素

  insert的返回值依赖于容器类型和参数。对不包含重复关键字 的容器，添加单一元素的insert和emplace版本返回一个pair,其中pair的first成员是一个迭代器，指向具有给定关键字的元素；second成员是一个bool值，指出元素是插入成功还是已经存在容器中。插入成果返回true，否则返回false。
- 关联容器的删除操作

c.erase(k)	**从c中删除每个关键字为k的元素。返回一个size_type值，指出删除的元素的数量**
c.erase(p)	从c中删除迭代器p指定的元素。p必须指向c中一个真实元素，不能等于c.end()。返回一个指向p之后元素的迭代器，若p指向c中的尾元素，则返回c.end()
c.erase(b,e)	删除迭代器对b和e所表示的范围中的元素，返回e

- map和unordered_map的下标操作

c[k]	**返回关键字为k的元素；如果k不在c中，添加一个关键字为k的元素，对其进行值初始化**
c.at(k)	返回关键字为k的元素，带参数检查；若 k不在c中，抛出一个out_of_range异常。

- 关联容器中查找元素的操作

  lower_bound和upper_bound不适用于无序容器

  下标和at操作只适用于非const的map和unordered_map

c.find(k)	**返回一个迭代器，指向第一个关键字为k的元素，若k不在容器中，则返回尾后迭代器**
c.count(k)	返回关键字等于k的元素的数量。对于不允许重复关键字的容器，返回值永远是0或1。
c.lower_bound(k)	返回一个迭代器，指向第一个关键字不小于k的元素
c.upper_bound(k)	返回一个迭代器，指向第一个关键字大于k的元素
c.equal_range(k)	返回 一个迭代器pair,表示关键字等于k的元素的范围。若k不存在，pair的两个成员军等于c.end()

- 无序容器得管理操作
  - 桶接口

c.bucket_count()	正在使用桶的数目
c.max_bucket_count()	容器能容纳的最多桶的数量
c.bucket_size(n)	第n个桶中有多少个元素
c.bucket(k)	关键字为k的元素在哪个桶中

- 桶迭代

local_iterator	可以用来访问桶中元素的迭代器类型
const_local_iterator	桶迭代器的const版本
c.begin(n),c.end(n)	桶n的首元素迭代器和尾后迭代器
c.cbegin(n),c.cend(n)	与前两个函数类似，但是返回const_local_iterator

- 哈希策略

c.load_factor()	每个桶的平均元素数量，返回float值
c.max_load_factor()	c试图维护的平均桶大小，返回float值。c会在需要时添加新的桶，已使得load_factor<=max_load_factor
c.rehash(n)	重组存储，使得bucket_count>=n且bucket_count>size/max_load_factor
c.reserve(n)	重组存储，使得c可以保存n个元素且不必rehash

动态内存与智能指针

shared_ptr 允许多个指针指向同一个对象

unique_ptr 则独占所指向的对象，即某一时刻只能有一个unique_ptr指向一个给定的对象。生命周期相绑定 。

weak_ptr 弱引用，指向shared_ptr 所管理 的对象；一种不控制所指向对象生存期的智能指针。

- 智能指针解引用 *p

析构函数

- 调用析构函数

  变量在离开其作用域时被销毁。

  当一个对象被销毁时，其成员被销毁。

  容器（无论时标准库容器还是数组）被销毁时，其元素被销毁。

  对于动态分配的对象，当对指向它的指针应用delete运算符时被销毁

  对于临时对象，当创建它的完整表达式结束时被销毁。

拷贝构造函数

=default 显示地要求编译器生成合成的版本

=delete 组织拷贝 注：析构函数不能是被删除的，否则无法销毁此类型的对象。

左值引用

变量 标准库move函数

标准库function

function类型能够表示对象的调用形式

function f; f是一个用来存储可调用对象的空function，这些可调用对象的调用形式应该与函数类型T相同。

面向对象

数据抽象 继承 动态绑定

动态绑定只有在virtual关键字情况下，才能够发生。

虚函数：派生类通过基类自定义自身的版本，此时基类就江函数声明虚函数。

继承关系中根节点的类通常都会定义一个虚析构函数。

C++中public、protected、private的区别

第一: private,public,protected的访问范围:

private: 只能由该类中的函数、其友元函数访问,不能被任何其他访问，该类的对象也不能访问.
protected: 可以被该类中的函数、子类的函数、以及其友元函数访问,但不能被该类的对象访问
public: 可以被该类中的函数、子类的函数、其友元函数访问,也可以由该类的对象访问
注：友元函数包括两种：设为友元的全局函数，设为友元类中的成员函数

第二:类的继承后方法属性变化:

使用private继承,父类的所有方法在子类中变为private;
使用protected继承,父类的protected和public方法在子类中变为protected,private方法不变;
使用public继承,父类中的方法属性不发生改变;

三种访问权限

public:可以被任意实体访问

protected:只允许子类及本类的成员函数访问

private:只允许本类的成员函数访问

- 派生类使用基类的构造函数来初始化它的基类部分。
- 派生类构造函数通过构造函数初始化列表来将实参传递给基类构造函数的。
- 在类名后面加final就能够放置继承
- 基类的指针或引用的静态类型可能与其动态类型不一致
- 不存在基类向派生类的隐式类型的转换
- 派生类向基类的自动类型转换只对指针或引用类型有效，在派生类型和基类类型之间不存在这样的转换

纯虚函数

在函数体的位置书写=0就能够将一个虚函数声明为纯虚函数。`double f()const = 0` 没有意义的函数

- 含有纯虚函数的类是抽象基类，不能实例化抽象基类。

友元

当一个类将另一个类声明为友元时，这种友元关系只对声明的类有效。对原来的那个类，其友元的基类或者派生类不具有特殊的访问能力。

默认继承

默认情况下，使用class关键字定义的派生类时私有继承的，而使用struct关键字定义的派生类时共有继承的。

- 声明在内层作用域的函数并不会重载声明在外层作用域的函数。定义派生类中的函数也不会重载其基类的成员。如果派生类的成员与基类的成员同名，则派生类将在其作用域内隐藏改该基类成员。

对Quote对象逐成员地进行拷贝、移动、赋值和销毁操作。

```
class Quote {
public:
    Quote() = default;                              //对成员依次进行默认初始化
    Quote(const Quote&) = default;                  //对成员依次拷贝
    Quote(Quote&&) = default;                       //对成员依次拷贝
    Quote& operator=(const Quote&) = default;       //拷贝赋值
    Quote& operator=(Quote&&) = default;            //移动赋值
    virtual ~Quote()=default;                   //对析构函数进行动态绑定
}
```

对象销毁顺序正好与其创建的顺序相反：派生类析构函数首先执行，然后是基类的析构函数。以此类推，沿着继承体系的反方向直至最后。

- 容器通过放置智能指针来实现对象的继承。

```
multiset<shares_ptr<Quote>,decltype(compare)*> itemms{compare};
```

上诉的含义 是一个指向Quote对象的shared_ptr的multiset。这个multiset将使用一个与compare成员类型相同的函数对其中的元素进行排序。multiset成员名字是items，初始化items并令其使用compare函数。

函数模板

```
tempplate <typename T>
int compare(const T &v1,const T &v2)
{
    if(v1<v2)  return -1;
    if(v2<v1) return 1;
    return 0;
}
```

函数模板可以声明为inline或constexpr

类模板

```
template<typename T> class Blob{}
```

引用折叠

- X& &、X& && 都折叠成类型X&
- 类型X&& &&折叠成X&&

左右值引用

右值引用T&

左值引用 T&&

std::move() 获得一个绑定到左值上的右值引用。

可变参数模板

```
template<typename T,typename... Args>
void foo(const T &t,const Args& ... rest)
```

sizeof...运算符 `sizeof...(Args)//类型参数的数目`

- 通过可变参数实现递归

```
//用来终止递归并打印最后一个元素的函数
//此函数必须在可变参数版本的print定义之前声明
template<typename T> ostream &print(ostream &os,const  T &t)
{
    return os<<t;
}
//包中除了最后一个元素之外的其他元素都会调用这个版本的print
template<typename T> ostream &print(ostream &os,const  T &t, const Args&...
rest)
{
     os<<t<<", ";
    //递归调用打印其他实参  rest的第一个实参被绑定到t，剩余实参形成下一个print调用的参数包
    return print(os,rest...);
}
```

bitset

```
    bitset<100> bst;        //定义100位长
    bst.set(10,true);       //将第10位设置为1
    cout<<bst.to_string()<<endl;
```

b.any()	b中是否存在置位的二进制位
b.all()	b中所有位都置位了吗
b.none()	b中不存在置位的二进制位吗
b.count()	b中置位的位数
b.size()	一个constexpr函数，返回b中的位数
b.set(pos,flag)	将pos位设置位flag（true/false）
b.set()	所有位置1
b.reset(pos)	将pos位置0
b.reset()	将所有位置0
b.flip(pos)	反转pos位
b.flip()	反转所有位
b[pos]	获得pos位的状态
b.to_string()	转换成字符串输出

正则表达式

regex	表示有一个正则表达式的类
regex_match	将一个字符序列与一个正则表达式匹配
regex_search	寻找第一个与正则表达式匹配的子序列
regex_replace	使用给定格式替换一个正则表达式
regex_iterator	迭代器适配器，调用regex_search来遍历一个string中所有匹配的子串
smatch	容器类，保存在string中搜索的结果
ssub_match	string中匹配的子表达式的结果

- demo

```
string s = "fnaji1565463f wq";
regex r("[a-z]*\\d+");
smatch result;
bool res = regex_search(s,result,r);
if(res){
    cout<<"result = "<<result.str()<<endl;
}
```

随机数

由随机引擎类 和 随机数分布类组成

引擎	类型，生成随机unsigned整数序列
分布	类型，使用引擎返回服从特定概率分布的随机数

- 随机数引擎操作

Engine e	默认构造函数；使用该引擎类型的默认的种子
Engine e(s)	使用整型值为s作为 种子
e.seed(s)	使用种子s重置引擎的状态
e.min()	此引擎可生成的最小值
e.max()	此引擎可生成的最大值
Engine::result_type	此引擎生成的unsigned整数类型
e.discard(u)	将引擎推进u步；u的类型为unsigned long long

IO流

控制布尔值的格式 boolalpha

指定进制 hex oct dec showbase/noshowbase(带格式输出） setbase(b)设置输出为b进制

设置浮点数精度 cout.setprecision(n) setprecision(n)