C++primer学习笔记(11–12章)

第十一章 关联容器

关联容器与顺序容器的不同之处：按关键字保存和访问

map 与 set

- 允许重复关键字的容器名字都包含 multi
- 不保持按照顺序存储的容器关键字以 unordered 开头

插入操作

- insert 时，若关键字已在容器中则什么也不做 下标插入时会修改值
- 对不允许重复 key 值的容器，insert 返回值的 first 是一个迭代器，指向具有关键字的元素，second 是一个 bool 值，指出元素是否插入成功
- 对允许重复 key 值的容器，insert 返回值对应指向元素的迭代器

删除操作

- erase 接受一个 key_type 参数，删除所有匹配给定关键字元素，返回实际删除的元素的元素

```C++
//使用事例
map<string,int> m;
set<string> s;
string str;
cin>>str;
//map插入方法
m[str]=1;
m.insert({str,1});
m.insert(make_pair(str,1));
m.insert(pair(<string,int>(str,1));
m.insert(map<string,int>::value_type(str,1));
//set插入方法
s.insert(str);
```

unordered_map(unordered_set)与 map(set)的区别

- unordered_map 底层是哈希表，map 底层是红黑树
- unordered_map 查找、插入、删除效率为 O(1)
- map 查找、插入、删除效率为 O(logN)
- unordered_map 会占据更大的空间 底层是开了一个大的数组

decltype 与 auto

- auto 根据等式右边推导类型，decltype 根据表达式推导类型
- auto 要求变量必须初始化

有序容器的关键字类型

- 比较所提供的操作必须在关键字上定义一个严格弱序
- 使用 decltype 来指出自定义操作类型

- set 的迭代器类型是 const 的 map 的 key 值是 const 的

```C++
bool cmp(const int &a,const int &b)
{
    return a>b;
}
map<int,int,decltype(cmp)*> m(cmp);
```

关联容器的额外类型别名

key_type	关键字类型
mapped_type	关键字关联的类型（只有相关 map 类型才定义了此类型）
value_type	相应的键值对（对于 set 来说，与 key_type 一样）

相关函数

- lower_bound 返回第一个指向关键字的元素
- upper_bound 指向最后一个匹配给定关键字元素之后的位置
- equal_range 接受一个关键字，返回一个迭代器 pair 第一个迭代器指向第一个与关键字匹配的元素，第二个指向最后一个匹配元素之后的位置

无序容器

- 在存储上组织成一个桶，每个桶保存一个或者多个元素
- 将具有一个特定哈希值或者相同关键字的保存在同一个桶中

无序容器相关操作

c.bucket_count()	正在使用的桶的数目
c.max_bucket_count()	容器能容纳的最多的桶的数目
c.buket_size(n)	第 n 个桶有多少个元素
c.bucket(k)	关键字为 k 的元素在哪个桶中
local_iterator	访问桶的迭代器类型
c.begin(n),c.end(n)	桶 n 的首尾迭代器

无序容器对关键字类型的要求

- 不能直接定义关键字类型为自定义类型的无序容器
- 需要重载哈希函值计算函数数与替代==运算符的函数

第十二章 动态内存

- 在堆上分配内存
- 运用 new 与 delete 运算符
- 一个动态分配的 const 对象必须进行初始化
- 智能指针负责自动释放所指对象 shared_ptr unique_ptr weak_ptr

shared_ptr 类

- 需指定类型
- 利用 make_shared 函数使用动态内存
- shared_ptr 都有一个引用计数器

- 当指向对象的最后一个 shared_ptr 被摧毁后，会自动销毁此对象
- 可以将 shared_ptr 与 new 结合使用，shared_ptr 拒绝隐式转换
- 将另一个指针指针绑定到 get 返回的对象是错误的，但是不会报错

```C++
shared_ptr<int> p=make_shared<int>(42);//创建智能指针
int n=p.use_count();//指针数
auto q(p);//拷贝
shared_ptr<int> p2(new int(1024));
int *ip;
ip=q.get();//得到一个内置指针
q.reset();//释放指针为空
```

指针值与 delete

```C++
int i,*pi1=&i,*pi2=nullptr;
double *pd = new double(33),*pd2=pd;
delete i;  //error. i is not a pointer
delete pi1;//error. pi1指向一个局部变量
delete pd;//right
delete pd2;//未定义 已释放
delete pi2;//right
```

unique_ptr 类

- 当定义一个 unique_ptr 时，需要将其绑定到一个 new 返回的指针上
- unique_ptr 不支持拷贝和赋值操作 但可以调用 release 或 reset 将指针所有权转移
- 调用 release 会切断 unique_ptr 和它管理对象之间的联系，所以常用来初始化另一个智能指针

```C++
unique_ptr<int> p1(new string("sss"));
unique_ptr<int> p2(p1);//error
int *p3;
p3.reset(p1);
```

weak_ptr 类

- 不控制指向对象生存期的智能指针，指向一个由 shared_ptr 管理的对象
- 一旦指向的被销毁，对象就会被释放
- 由于对象可能不存在，不能直接使用 weak_ptr 访问对象，必须调用 lock,
- 如果存在，lock 返回一个指向共享对象的 shared_ptr

```C++
auto p=make_shared<int>(42);
weak_ptr<int> wp(p);
if(shared_ptr<int> np=wp.lock())
{
    ...................
}
```

动态数组

- 动态数组并不是数组类型，而是得到一个数组类型的指针
- 不能对动态数组调用 begin 和 end
- 可以创建大小为 0 的动态数组

```C++
```

```
//分配事例
typedef int arr[20];
int *pia=new int[20];//20个未初始化的int
int *pia2=new int[20]();//10个初始值为0的int
int *pia2=new int[20]{0,1,2};
int *pia3=new arr;
delete []pia;
.....................
```

allocator 类

- 将内存分配与对象构造分离开来
- 根据对象类型来确定恰当的内存大小与对齐位置
- 必须对每个构造的元素调用 destory 来摧毁他们

C++

```
allocator<string> alloc;//可以分配string的对象
auto const p=alloc.allocate(n);//分配n个未初始化的string对象
auto q=p;//q指向最后构造元素之后的位置
alloc.construct(q++);//构造
```