关联容器：

第19条：

相等：基于operator==，如果“x == y”返回真，则x和y相等。

等价：x和y的任何一个（按照一定的排序准则）都不在另一个的前面，那么x和y（按照这一准则）就是等价的，即：!(x<y) && !(y<x)为true。

关联容器是通过等价来保持排列顺序的。

第20条：

为包含指针的关联容器指定比较类型。

例：程序effective\_stl\_test7

struct StringPtrLess :

public *binary\_function*<const *string*\*, const *string*\*, bool>

{

bool operator ()(const *string*\* ps1, const *string*\* ps2) const

{

return \*ps1 < \*ps2;

}

};

// 第2个参数是指定的指针的比较类型

*set*<*string*\*, StringPtrLess> string\_set;

// 如果未指定比较类型，则会按照指针地址的大小排序

//set<string\*> string\_set;

string\_set.*insert*(new *string*("Anteater"));

string\_set.*insert*(new *string*("Wombat"));

string\_set.*insert*(new *string*("Lemur"));

string\_set.*insert*(new *string*("Penguin"));

for (auto i = string\_set.*begin*(); i != string\_set.*end*(); ++i)

{

*cout* << \*\*i << *endl*;

}

输出为：

Anteater

Lemur

Penguin

Wombat

可以针对指针的比较类型实现一个更为通用的形式：

// 更为通用的指针比较类型

struct DereferenceLess

{

template<typename PtrType>

bool operator() (PtrType pt1, PtrType pt2) const

{

return \*pt1 < \*pt2;

}

};

set<string\*, DereferenceLess> string\_set;

第21条：

针对关联容器set和map，保证比较函数在等价值情况下返回false。

对于set和map，不允许有等价中或key等价的值，插入时如果存在，就会被自动忽略掉。但如果指定的比较函数在等价值情况下返回true，就会引发无法预料的后果，在VS2017下会抛出异常。

例：程序effective\_stl\_test8

// 错误的例子

*set*<int, *less\_equal*<int>> set1;

set1.*insert*(10);

// error，抛出异常，比较函数在等值情况下返回true

// set1.insert(10);

*map*<int, int, *less\_equal*<int>> map1;

map1.*insert*({1, 2});

// error，抛出异常

//map1.insert(make\_pair(1, 3));

第22条：

关联容器，set和multiset不要直接修改其元素值，如果希望修改，可以先删除其元素，然后插入新的元素值，以保持容器的顺序。map和multimap不能直接修改其key值，会报编译错误，如果希望更新key值，也需要先删除某个key值，然后插入新的key/value。

第23条：

尽量使用排序的vector来取代关联容器，大多数时候其效率更高，占用空间更少。

第24条：

map容器中的operator[]和insert对效率的影响。

添加map元素时：

例：程序effective\_stl\_test9

class Widget

{

public:

Widget()

{

*cout* << "Default Construct Widget " << *hex* << this << *endl*;

}

Widget(double weight)

{

*cout* << "Construct Widget with double param " << *hex* << this << *endl*;

}

Widget(const Widget& org)

{

*cout* << "Copy construct Widget " << *hex* << this << " from " << &org << *endl*;

}

~Widget()

{

*cout* << "Destruct Widget " << *hex* << this << *endl*;

}

Widget& operator=(const Widget& org)

{

*cout* << "Operator= Widget " << *hex* << this << " from " << &org << *endl*;

return \*this;

}

};

typedef *map*<int, Widget> IntWidgetMap;

*map*<int, Widget> map1;

*cout* << "operator[] add element" << *endl*;

map1[1] = 1.5;

*cout* << "operator[] end" << *endl*;

*cout* << *endl*;

*cout* << "insert() add element" << *endl*;

map1.*insert*(IntWidgetMap::*value\_type*(2, 1.8));

*cout* << "insert() end" <<*endl*;

*cout* << *endl*;

输出为：

operator[] add element

Construct Widget with double param 0094FB8B

Default Construct Widget 009A89A4

Operator= Widget 009A89A4 from 0094FB8B

Destruct Widget 0094FB8B

operator[] end

insert() add element

Construct Widget with double param 0094FB7C

Copy construct Widget 009B1014 from 0094FB7C

Destruct Widget 0094FB7C

insert() end

Destruct Widget 009B1014

Destruct Widget 009A89A4

通过operator[]插入元素时，operator[]返回引用，指向与key=1关联的对象，如果该对象不存在，使用默认构造函数创建一个对象（009A89A4）并指向它。然后通过operator=将之前构造的临时对象Widget(1.5)（0094FB8B）赋给该对象，最后析构掉0094FB8B临时对象。

通过insert插入元素，创建临时对象Widget(1.8)（0094FB7C），然后拷贝给一个对象（009B1014）并插入map容器。

从效率上讲，插入时，insert比operator[]更高效。而且operator[]在没有key值存在时，自动插入的功能需要谨慎使用，避免插入并非需要的key值。

更新元素时：

例：程序effective\_stl\_test9

*cout* << "operator[] update element" << *endl*;

map1[1] = 2.5;

*cout* << "operator[] end" << *endl*;

*cout* << *endl*;

输出为：

operator[] update element

Construct Widget with double param 00F6FC33

Operator= Widget 012D89D4 from 00F6FC33

Destruct Widget 00F6FC33

operator[] end

使用insert更新元素，其实是先删除之前的key，然后再重新插入新的key/value，效率较operator[]要低。

map容器，operatr[]更新效率高，insert插入效率高。

高效的插入或更新map容器：

例：程序effective\_stl\_test9

// 更加有效的插入或更新

template<typename MapType,

typename KeyArgType,

typename ValueArgType>

typename MapType::*iterator*

EffectiveAddOrUpdate(MapType& m, const KeyArgType& k, const ValueArgType& v)

{

// 使用typename避免语法的二义性

// 指明MapType::iterator是类型，而

// 不是静态变量

// 查找第一个key值大于或等于k的元素并返回其迭代器

typename MapType::*iterator* lb = m.*lower\_bound*(k);

// 键值k存在

if (lb != m.*end*() &&

!(m.*key\_comp*()(k, lb->*first*)))

{

lb->*second* = v;

return lb;

}

else // 键值k不存在

{

typedef typename MapType::*value\_type* MVT;

return m.*insert*(lb, MVT(k, v));

}

}