连续内存容器：基于数组的容器，有vector，deque，string

基于节点的容器：list，关联容器，无序容器（哈希容器）

第1条：

容器类型的选择：

1. 是否需要在容器的任意位置插入新元素？如果需要，就选择序列容器，关联容器是不行的。
2. 是否关心容器中的元素是如何排序的？如果不关心，可以考虑哈希容器；否则，不能选择哈希容器。
3. 需要哪种类型的迭代器？如果必须是随机访问迭代器，则只能选择vector，deque和string。
4. 发生元素的插入或删除操作时，是否要避免移动容器中原来的元素？如果是，避免选择连续内存的容器。
5. 容器中数据的布局是否需要和C兼容？如果需要兼容，就只能选择vector。
6. 元素的查找速度是否是关键的考虑因素？如果是就考虑哈希容器、排序的vector和关联容器。

第2条：

不要试图编写独立于容器类型的代码，试图编写对序列容器和关联容器都适用的代码毫无意义。

第3条：

STL容器保存的对象通常都是指定对象的拷贝，从STL容器中取出的对象也是容器中保存的对象的拷贝。

例：程序effective\_stl\_test1

class Base

{

public:

Base()

{

*cout* << "Construct Base" << *endl*;

}

Base(const Base& org)

{

*cout* << "Copy construct Base" << *endl*;

}

virtual ~Base()

{

*cout* << "Destruct Base" << *endl*;

}

virtual void Print()

{

*cout* << "Print Base" << *endl*;

}

};

*cout* << "Original Object" << *endl*;

Base b;

*vector*<Base> base\_vector;

*cout* << *endl*;

*cout* << "Push in vector" << *endl*;

base\_vector.*push\_back*(b);

*cout* << *endl*;

*cout* << "Get from vector" << *endl*;

Base b1 = base\_vector[0];

输出为：

Original Object

Construct Base

Push in vector

Copy construct Base

Get from vector

Copy construct Base

Destruct Base

Destruct Base

Destruct Base

在此基础上，要确保容器中的对象拷贝正确而高效。

存在继承关系时，不要把派生类的对象插入基类的容器中。拷贝动作会进行剥离，派生类对象通过基类的拷贝构造函数进入基类，自身特有的信息会丢失，也不会触发虚函数机制。

例：程序effective\_stl\_test1

class Derived : public Base

{

public:

Derived()

{

*cout* << "Construct Derived" << *endl*;

}

Derived(const Derived& org) : Base(org)

{

*cout* << "Copy construct Derived" << *endl*;

}

~Derived()

{

*cout* << "Destruct Derived" << *endl*;

}

void Print() override

{

*cout* << "Print Derived" << *endl*;

}

};

*cout* << "Original Derived Object" << *endl*;

Derived d;

*vector*<Base> base\_vector1;

*cout* << *endl*;

*cout* << "Push in Base vector" << *endl*;

base\_vector1.*push\_back*(d);

*cout* << *endl*;

*cout* << "Call virtual function" << *endl*;

base\_vector1[0].Print();

*cout* << *endl*;

输出为：

Original Derived Object

Construct Base

Construct Derived

Push in Base vector

Copy construct Base

Call virtual function

Print Base

Destruct Base

Destruct Derived

Destruct Base

从输出可以看出，派生类对象插入基类容器时，只调用了基类的拷贝构造函数，而且容器中的对象调用虚函数Print()时调用的是基类的函数。

第4条：

调用empty而不是检查size()是否为0来判断容器是否为空。

empty()对于所有的标准容器都是常数时间操作，size()对于list要耗费线性时间。

第5条：

STL容器对于容器的创建、插入、删除和赋值一般都会有区间成员函数和对应的单元素成员函数两种。使用区间成员函数要优先于与之对应的单元素成员函数。

给定的两个vector容器v1和v2，是v1的内容和v2的后半部分相同，使用

v1.assign(v2.begin() + v2.size() / 2, v2.end());

是最简单的操作，要避免循环使用对应的单元素函数。

第6条：

C++编译器的分析机制，会尽可能的把语句解释为函数的声明。

例：程序effective\_stl\_test2

class Test

{

public:

Test()

{

*cout* << "Construct Test" << *endl*;

}

};

// 声明了一个函数，名称为t，返回值为Test

Test t();

对于STL容器，也会出现类似的情况：

例：程序effective\_stl\_test2

*ifstream* data\_file("test.txt");

// data是一个函数，返回值为list<int>

// 第一个参数为data\_file，类型为istream\_iterator<int>

// 第二个参数没有名称，它是一个函数指针，指向一个

// 返回istream\_iterator<int>的函数

*list*<int> data(*istream\_iterator*<int>(data\_file), *istream\_iterator*<int>());

在使用STL容器，要注意避免这种操作。

例：程序effective\_stl\_test2

*istream\_iterator*<int> data\_begin(data\_file);

*istream\_iterator*<int> data\_end;

*list*<int> data(data\_begin, data\_end);

for (auto v : data)

{

*cout* << v << *endl*;

}