Standard Template Library

3th Edition

如果说我看的远，那是因为我站在巨人的肩上。

Auth : 王桂林

Mail : guilin_wang@163.com

Org : 能众软件

Web : http://edu.nzhsoft.cn

版本信息:

版本	修订人	审阅人	时间	组织
v1.0	王桂林		2016.04.02	
v2.0	王桂林		2017.06.01	山东能众软件科技有限公司
v3.0	王桂林		2018.12.08	山东能众软件科技有限公司

更多资料:

1. STL 综述

1.1.STL 是什么

STL = Standard Template Library，首先他是一个 Library，也就是一个函数库,就像大家以前用的函数 sin/random 等来自数学库，printf/fopen/fread/fwrite 等来自 io 库，strcpy/strcmp/strcat 等来自己字符串库。

每一种函数库都解决了一类问题。比如数学库，解决通用数据运算问题，io 库解决了输入输出问题， 字符串库解决了字符串的生成，复制，拼接等等问题，避免了重复造轮子，提高了开发效率，STL 作为一个库，解决了哪些问题呢，就是很多入门的学生很头疼的问题，数据结构与算法。从此呢，数据结构和算法，不再是王谢堂前燕，经过 STL 后，已飞入寻常百姓家了。

其次，就是 Template,中文是模板的意思， 模板解决了什么问题呢， 泛型编程， 泛型编程最早也是一种理念，简单的来讲就是用一种模子，解决一类问题。C++引入 Template 后，泛型编程才算是落到了实处，其实也是解决避免重复造轮子，提高开发效率的问题。

再者就是，Standard 标准。为什么说是标准呢，已经被 C++委员会纳入 C++标准的一部分。以前说 C++有三大特征，封装，继承，多态。现在至少要说，C++四大特征 封装，继承，多态和 STL。

综上所述，STL 是由 C++模板编写的一套已纳入 c++标准的类库，该库解决了通用数据结构与算法的问题。

1.2.STL 作者

STL 它是由 Alexander Stepanov、Meng Lee 和 David R Musser 在惠普实验室工作时所开发出来的。

亚历山大·斯特潘诺夫(AlexanderStepanov),STL(标准模板库)之父，并因此而荣获第一届 Dr. Dobb's 程序设计杰出奖， 现在是 Adobe 公司首席科学家。

程序基于精确的数学——STL 之父 Alex Stepanov。

https://en.wikipedia.org/wiki/Alexander_Stepanov

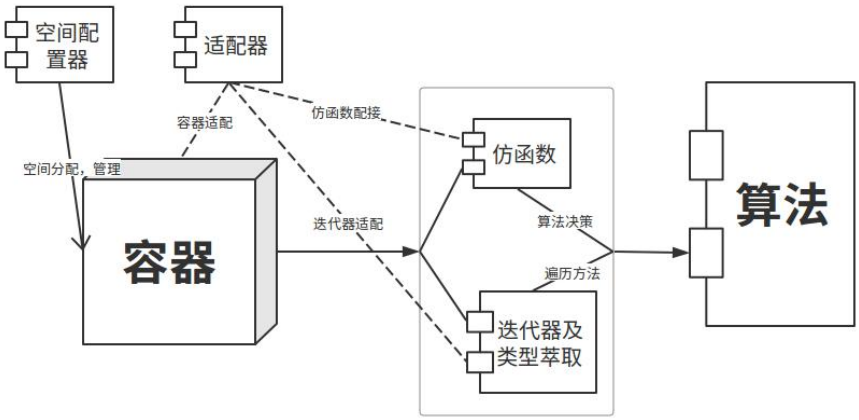
他曾是康柏电脑公司的副总裁和首席科学家，AT&T 实验室副总裁和首席架构师，SGI 服务和超级计算机业务首席技术官。

1.3.STL Category

1.3.1. 组成图示

STL 主要由空间适配器 allocator，容器 container，算法 algorithm,迭代器 iterator 和

仿函数 functor

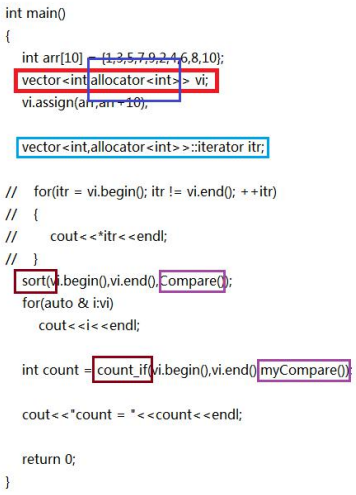


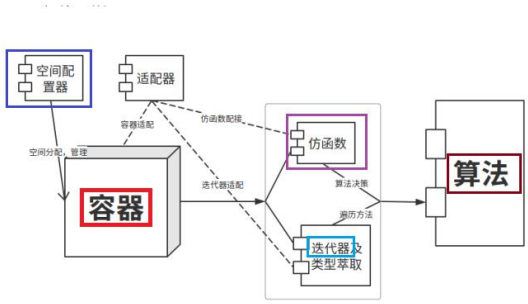
1.3.2. 示例

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

class Compare
{
public:
    bool operator()(int x, int y)
    {
        return x<y;
    }
};

int main()
{
    int arr[10] = {1,3,5,6,7,2,4,6,8,10};
    vector<int,allocator<int>> vi(arr,arr+10);
    vector<int>::iterator itr;
    for(itr = vi.begin(); itr != vi.end(); itr++)
    {
        cout<<*itr<<endl;
    }
    sort(vi.begin(),vi.end(),Compare());

    for(auto &i:vi)
        cout<<i<<endl;
```

```
    int count =
            count_if(vi.begin(),vi.end(),
                    bind(greater<int>(),placeholders::_1,2));
//          count_if(vi.begin(),vi.end(),bind2nd(greater<int>(),2));
    cout<<"count = "<<count<<endl;


    return 0;
}
```

1.3.3. 代码图示对照





1.4.why STL

●高可重用性:

STL 中几乎所有的代码都采用了模板类和模版函数的方式实现，这相比于传统的由函数和类组成的库来说提供了更好的代码重用机会。

●高性能:

如 map 可以高效地从十万条记录里面查找出指定的记录,因为 map 是采用红黑树的变体实现的。 (红黑树是平横二叉树的一种)

●高移植性:

如在项目 A 上用 STL 编写的模块，可以直接移植到项目 B 上。

●跨平台:

如用 windows 的 Visual Studio 编写的代码可以在 Mac OS 的 XCode 上直接编译

1.5.STL 版本及源码下载

1.5.1. 版本

ANSI/ISO 的 C++ STL 规范版本正式通过以后，各个 C++编译器厂商就可以依照标准所

描述的原型去实现 C++ STL 泛型库，于是出现多种符合标准接口，但具体实现代码不同的泛型库，主要有：

●HP STL

HP STL 是 Alexandar Stepanov 在惠普 Palo Alto 实验室工作时，与 Meng Lee 合作完成的。HP STL 是 C++ STL 的第一个实现版本，而且是开放源码。其它版本的 C++ STL 一般是以 HP STL 为蓝本实现出来的。

●STLport

为了使 SGI STL 的基本代码都适用于 VC++和 C++ Builder 等多种编译器，俄国人 Boris Fomitchev 建立了一个 free 项目来开发 STLport，此版本 STL 是开放源码的。

●P.J.Plauger STL

由 P.J.Plauger 参照 HP STL 实现出来，被 Visual C++编译器所采用，但不是开源的。

●Rouge Wave STL

由 Rouge Wave 公司参照 HP STL 实现，用于 Borland C++编译器中，这个版本的 STL 也不是开源的。

●SGI STL

由 Silicon Graphics Computer Systems 公司参照 HP STL 实现，主要设计者仍然是 STL 之父 Alexandar Stepanov，被 Linux 的 C++编译器 GCC 所采用。SGI STL 是开源软件，源码可读性甚高。

1.5.2. 源码

http://www.martinbroadhurst.com/stl/

2. 模板 Template

2.1.起源

2.1.1. 宏函数

静态类型的语言，满足了高效性和安全性，但为提供通用型代码提供了障碍，比如，操作一组同一逻辑的不同类型数据，需要写除了类型不同以外的几乎相同的代码。

```
int max_int(int x, int y)            {return x<y? y:x;}
double max_double(double x, double y ){return x<y? y:x;}
string max_string(string x, string y ){return x<y? y:x;}
```

在 C 语言时代，就开始研究，如何写通用型代码了。典型的就是宏函数，宏函数发生在预处理阶段，纯文本替换，会带来类型不安全的问题。

```
#include <iostream>
using namespace std;

//int max_int(int x, int y) {return x<y? y:x;}
//double max_double(double x, double y ){return x<y? y:x;}
//string max_string(string x, string y ){return x<y? y:x;}


#define  max_int(x,y)    ((x)<(y)? (y):(x))
#define  max_double(x,y)  ((x)<(y)? (y):(x))
#define  max_string(x,y)  ((x)<(y)? (y):(x))

int main()
{
    cout<<"max 4,5 = " << max_int(4,5)<<endl;
    cout<<"max 4.5,5.5 = "<<max_double(4.5,5.5)<<endl;
    cout<<"max china,nzhsoft = "<<max_string("china","nzhsoft")<<endl;

    return 0;
}
```

该段程序，如果用 g++编译，是没有问题的，如果采用 gcc 编译，则会出问题,问题就出在 char*上，导致语义不明确。

```
#include <stdio.h>

//#define  max_int(x,y)    ((x)<(y)? (y):(x))
//#define  max_double(x,y)  ((x)<(y)? (y):(x))
//#define  max_string(x,y)  ((x)<(y)? (y):(x))
```

```
#define max(x,y)  ((x)<(y)? (y):(x))

int main()
{
   printf("max 4,5 = %d",max(4,5));
   printf("max 4.5,5.5 = %f ",max(4.5,5.5));
   printf("max china,nzhsoft = %p ",max("china","nzhsoft"));
    return 0;
}
```

2.1.2. 宏函数替换类型

如果用宏函数来替换类型，会如何呢？也就是利用了宏的一般性与类型的安全性相结合。

```
#define   MAX(T)  T max_##T(T x,T y) {return x<y? y:x;}
MAX(int) MAX(double) MAX(string)

int main()
{
    cout<<"max 4,5 = " << max_int(4,5)<<endl;
    cout<<"max 4.5,5.5 = "<<max_double(4.5,5.5)<<endl;
    cout<<"max china,nzhsoft = "<<max_string("china","nzhsoft")<<endl;

    return 0;
}
```

核心思想是，两次替换，一次是类型，另外一次是参数。可以再加一个宏，使其更简洁：
#define max(T) max_##T

```
#include <iostream>
using namespace std;

//int max_int(int x, int y) {return x<y? y:x;}
//double max_double(double x, double y ){return x<y? y:x;}
//string max_string(string x, string y ){return x<y? y:x;}


#define   MAX(T)  T max_##T(T x,T y) {return x<y? y:x;}
MAX(int) MAX(double) MAX(string)
#define max(T) max_##T

int main()
{
    cout<<"max 4,5 = " << max(int)(4,5)<<endl;
    cout<<"max 4.5,5.5 = "<<max(double)(4.5,5.5)<<endl;
    cout<<"max china,nzhsoft = "<<max(string)("china","nzhsoft")<<endl;

    return 0;
```

```
}
```

2.2.函数模板

既然是预处理器阶段都可以帮我们完成类型与参数的替换,实现了通常代码的编写与类型安全。用编译器能不能实现呢？，编译器实现的版本，就是模板。

2.2.1. 定义

模板，简言之，即类型参数化，使抽象函数(通用函数)成为了可能。将泛型编程推到了极致。

上例中的代码，可由模板简写为:

```
template<class T>    //class 或 typename
T Max (T x ,T y )
{
    return x <y ? y:x;
}

int main()
{
    cout<<"max 4,5 = " << Max<int>(4,5)<<endl;  //::max 也可以解决命名冲突
    cout<<"max 4.5,5.5 = "<<Max<double>(4.5,5.5)<<endl;
    cout<<"max china,nzhsoft = "<<Max<string>("china","nzhsoft")<<endl;

    return 0;
}
```

2.2.2. 函数生成

模板并不会生成处理任何类型的实体,而是对于实例化模板参数的类型都从模板产生一个不同的实体，这种具体类型代替模板参数的过程称为模板实例化。实例化后的函数构成了重载关系，这种重载关系由C++类处理。

nm a.out

```
0000000000201d78 t __init_array_end
0000000000201d68 t __init_array_start
0000000000000ba0 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
0000000000000b90 T __libc_csu_fini
0000000000000b20 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
000000000000095a T main
00000000000008c0 t register_tm_clones
0000000000000850 T _start
0000000000202010 D __TMC_END__
```

```
0000000000000aca W _Z3MaxIdET_S0_S0_
0000000000000aae W _Z3MaxIiET_S0_S0_
0000000000000af2 W _Z3MaxIPKcET_S2_S2_
0000000000000a50 t _Z41__static_initialization_and_destruction_0ii
                 U _ZNSolsEd@@GLIBCXX_3.4
                 U _ZNSolsEi@@GLIBCXX_3.4
```

2.2.3. .hpp

每个函数模板事实都被编译了两次，第一次是在实例化这前，先检查模板代码本身，查看与类型无关代码的合法性。第二次，发生在实例化期间，结合所有的类型参数，再次检查模板代码，生成二进制代码。

第一次编译，不会生成二进制，而是在编译器内部生成了描述该函数模板的数据结构，所谓模板的内部表示。

当使用函数模板并引发实例化的过程中，编译器需要查看模板的定义（而普通函数只需要声明即可）。鉴于此，通常将模板的声明与实现写同一个文件中，以（.cpp）结尾。

2.2.4. 隐式推断

2.2.4.1. 引入-推断失败

在某些情况下，确实是可以采用隐式推断的方式，来达到简写的目的。

如果函数模板的调用参数的类型，相关于该模板的模板参数，编译器是有能力依据调用参数来推断模板参数，以获得与普通函数调用一致的语法。但是要承担的是隐式推断失败的情况 Max("china","nzhsoft")。

```
template<class T>
T Max (T x ,T y )
{
   return x <y ? y:x;
}

int main()
{
   cout<<"max 4,5 = " << Max<int>(4,5)<<endl;
   cout<<"max 4.5,5.5 = "<<Max<double>(4.5,5.5)<<endl;
   cout<<"max china,nzhsoft = "<<Max("china","nzhsoft")<<endl;

   return 0;
}
```

推断逻辑是比较复杂的，我们不从原理上进行推理，仅就个别问题进行测试。避免隐式推断失败的最好办法，就是不用隐式推断。

2.2.4.2. 隐式推断常见规则

- 返回值的类型不参与隐式推断
- 隐式推断的同时，不允许隐式类型转换
- 不是全部模板参数都与调用参数的类型相关

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>

using namespace std;

template <typename T>
void foo( T const& x,  T const & y)
{
    cout<<typeid (x).name()<<" "<<typeid (y).name()<<endl;
}

template <typename T>
void func(T & x, T & y)
{
    cout<<typeid (x).name()<<" "<<typeid (y).name()<<endl;
}

template <typename R,typename T>
R bar(T & t)
{
    R r;
    cout<<typeid (r).name()<<" "<<typeid (t).name()<<endl;
    return r;
}


int main()
{
//    int a, b;
    double a,b;
    foo(a,b);
    func(a,b);

    char s[256],s2[256];
    foo(s,s2);
    func(s,s2);

//    foo("china","nzhsoft");
//    func("china","nzhsoft");
```

```
    int m; double n;

//    m =bar(n);
    m = bar<int>(n);

//    foo(m,n);
    foo<int>(m,n);

    return 0;
}

//模板参数与调用参数不匹配。
template<typename A,typename V> void foo(A arg){...V var...}
foo(1);
```

2.2.4.3. 半隐半显

函数模板的参数，可以部分显式指定，部分隐式指定，原则：必须显式指定最后一个不能被隐式推断的模板参数及其之前的所有模板参数。

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

template<typename R,typename V,typename A,typename B>
R foo(A a,B b ){
    V var;
    R r;
    return r;
}

int main()
{
    foo<int,string,double,char>(1.23,'a');
    foo<int,string,double>(1.23,'a');
    foo<int,string>(1.23,'a');
    foo<int>(1.23,'a');      //导致编译错误，string是最后一个不能隐式推断的
}
```

2.2.5. 缺省参数

2.2.5.1. 示例

```
#include <iostream>
#include <cstdlib>
```

```
#include <typeinfo>
using namespace std;

template<typename T = int>
T max(T a,T b)
{
    return a>b? a:b;
}

int main()
{
    cout<<::max(1,2)<<endl;
    cout<<::max(2.2,3.4)<<endl;
}
```

2.2.5.2. 隐式与缺省

默认参数与函数重载，有时会带来编译器的抱怨。优先选择默认，是我们最佳的选择。

关于模板中的默认参数(缺省)和隐式推断，C++2011 标准，允许函数模板带有缺省模板参数，但是依然会优先选择隐式推断的结果。

2.2.6. 模板函数特化

2.2.6.1. equal

```
#include <iostream>
#include <cstring>
using namespace  std;

template<typename T>
bool equal(T x, T y)
{
    return x == y;
}

template<>
bool equal(const char* x, const char* y)
{
    return strcmp(x,y) == 0;
}
int main()
{
    int a = 1 , b = 1;
    cout<<equal(a,b)<<endl;

    string aa = "nzhsoft", bb = "nzhsoft";
    cout<<equal(aa,bb)<<endl;
```

```
    char pa[] = "nzhsoft"; char pb[] = "nzhsoft";
    cout<<equal<const char *>(pa,pb)<<endl;

    return 0;
}
```

2.3.类模板

2.3.1. 常规案例

2.3.1.1. 引入 Compare

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

template<typename T>
class Compare{
public:
    Compare(T x, T y){
        _x = x;
        _y = y;
    }
    T max();
//    T max(){
//        return _x>_y? _x:_y;
//    }

private:
    T _x;
    T _y;
};

template<typename T>
T Compare<T>::max(){
    return _x>_y? _x:_y;
}

int main()
{
    Compare<int>  comp(3,5);
    cout<<comp.max()<<endl;
    return 0;
}
```

2.3.1.2. Stack<T>

2.3.1.3. Queue<T>

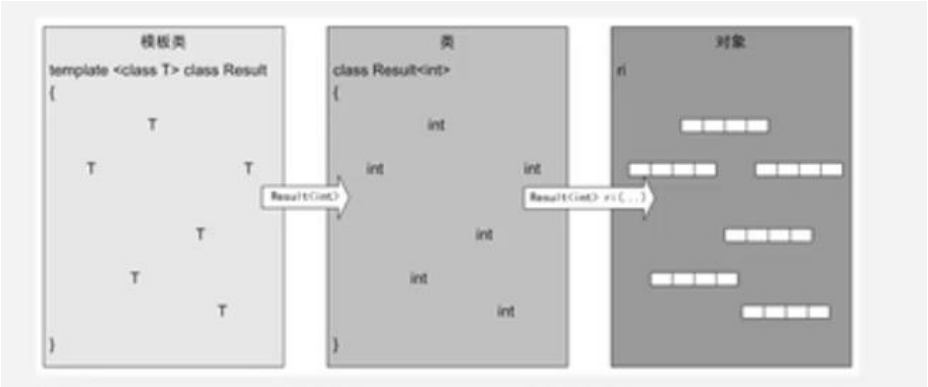
2.3.2. 实例化

2.3.2.1. class<T> ->class<int>

这个阶段发生在编译期，编译器将模板实例化生成对象创建类。

2.3.2.2. class<int> v

处理器执行对象创建指令将类实例化为对象。



2.3.2.3. 模板类型

在类模板中，但凡是表示类型的场合，都应该采用类模板名<模板参数>，但是多数编译器允许简化为类模板名。

```
template<typename T>
class Compare{
public:
    Compare(T x, T y){
        _x = x;
        _y = y;
    }
    Compare(const Compare<T> & another)  // Compare 也是可以的
    {
        _x = another._x;
        _y = another._y;
    }
    T max();
//    T max(){
//        return _x>_y? _x:_y;
//    }

private:
    T _x;
```

```
    T _y;
};
```

2.4.类模板特化

当类模板的类型参数取某些特定类型时，导致语义上的差别，这种差别，可能偏离了本意，怎么解决呢，特化。

如果可能，尽量不要用特化。

2.4.1. 二义来源

以 Compare 为例，当实例化类型分别为 string 和 char*的时候，我们想表达的是同一个意思，但编译运行的结果却是不同的。

```
#include <iostream>
#include <cstdlib>
#include <typeinfo>
using namespace std;

template<typename T>
class Compare{
public:
    Compare(T x, T y){
        _x = x;
        _y = y;
    }
    Compare(const Compare<T> & another)
    {
        _x = another._x;
        _y = another._y;
    }
    T max();
//    T max(){
//        return _x>_y? _x:_y;
//    }

private:
    T _x;
    T _y;
};

template<typename T>
T Compare<T>::max(){
    return _x>_y? _x:_y;
}

int main()
```

```
{
    Compare<string> cs("china","nzhsoft");
    cout<<cs.max()<<endl;

    Compare<char*> cs2("china","nzhsoft");
    cout<<cs2.max()<<endl;
    return 0;
}
```

2.4.2. 全特化(整类特化)

2.4.2.1. 语法

以 template<>开始，原通用模板中的 T 全部用特化类型来代替(const char*)。这样的类模板，就构成了全特化。

```
template<typename T>
class Compare{

template<>
class Compare<const char*>{
```

```
Compare(T x, T y){
Compare(const char* x, const char* y){
```

```
const char* Compare<const char*>::max()
{
    if(strcmp(_x,_y)>0)
        return _x;
    else
        return _y;
}
```

2.4.2.2. 实战

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <typeinfo>
using namespace std;

template<typename T>
class Compare{
public:
    Compare(T x, T y){
```

```
        _x = x;
        _y = y;
    }
    Compare(const Compare<T> & another)
    {
        _x = another._x;
        _y = another._y;
    }
    T max(){
        return _x>_y? _x:_y;
    }

private:
    T _x;
    T _y;
};

template<>
class Compare<const char*>{
public:
    Compare(const char* x, const char* y){
        _x = x;
        _y = y;
    }
    Compare(const Compare<const char*> & another)
    {
        _x = another._x;
        _y = another._y;
    }
    const char* max(){
        if(strcmp(_x,_y)>0)
            return _x;
        else
            return _y;
    }

private:
    const char* _x;
    const char* _y;
};


int main()
{
    Compare<string> cs("china","nzhsoft");
    cout<<cs.max()<<endl;

    Compare<const char*> cs2("china","nzhsoft");
```

```
    cout<<cs2.max()<<endl;
    return 0;
}
```

全特化，类外实现:

```
template<>
class Compare<const char*>{
public:
    Compare(const char* x, const char* y);
    Compare(const Compare<const char*> & another);
    const char* max();
private:
    const char* _x;
    const char* _y;
};

Compare<const char*>::Compare(const char* x, const char* y)
{
    _x = x;
    _y = y;
}

Compare<const char*>::Compare(const Compare<const char*> & another)
{
    _x = another._x;
    _y = another._y;
}

const char* Compare<const char*>::max()
{
    if(strcmp(_x,_y)>0)
        return _x;
    else
        return _y;
}
```

2.4.3. 局部特化(部分模板参数)

```
template<typename T , char>
```

2.4.4. 局部特化-偏特化(修饰)

全特化的版本，成本太高，类模板除了可以整体进行特化以外，也可以只针对部分成员函数进行特化。

针对整个类模板的特化，相当于定义一个全新的类，它的实现除了**类名**以外可以和基本版本完全不同。

针对类模板成员的函数特化则只是给出一个新的定义,该定义与其基本实现**共享同一个函数声明。**

2.4.4.1. 语法

以 template<>开始，原通用模板中特化函数中的 T 全部用特化类型来代替(const char*)。这样的局部替换，就构成了偏特化。

```
template<>
const char * Compare<const char *>::max(){
    if(strcmp(_x,_y)>0)
        return _x;
    else
        return _y;
}
```

2.4.4.2. 实现

```
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <typeinfo>
using namespace std;

template<typename T>
class Compare{
public:
    Compare(T x, T y){
        _x = x;
        _y = y;
    }
    Compare(const Compare<T> & another)
    {
        _x = another._x;
        _y = another._y;
    }
    T max(){
        return _x>_y? _x:_y;
    }

private:
    T _x;
    T _y;
};
template<>
const char * Compare<const char *>::max(){
    if(strcmp(_x,_y)>0)
        return _x;
    else
```

```
        return _y;
}

int main()
{
    Compare<string> cs("china","nzhsoft");
    cout<<cs.max()<<endl;

    Compare<const char*> cs2("china","nzhsoft");
    cout<<cs2.max()<<endl;
    return 0;
}
```

3. 容器 Container

3.1.容器

容器者，存储数据之所也。在泛型的基础上，设计的用于数据存储，运算，表示的模型，称之为容器。

3.2.特点 feature

3.2.1. 支持泛型

```
int main()
{
    vector<int> vi;
    vector<double> vd;
    vector<string>  vs;

    list<int> li;
    list<double> ld;
    list<string>  ls;

    map<int,string> mis;
    map<string,int> msi;
    return 0;

}
```

3.2.2. 保存副本

```
class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<"from"<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<"from"<<&other<<endl;
```

```
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};

int main()
{
    vector<A>  va;
    va.push_back(A());

    return 0;
}
```

3.2.3. 内存托管

容器采用了内存的托管机制，也就是说，放入容器中的对象，对象内存由容器来统一管理，我们只需要使用容器即可。

但如果托管的是指针，容器只负责，指针本身的内存大小，而其指向的空间还要作单独处理。

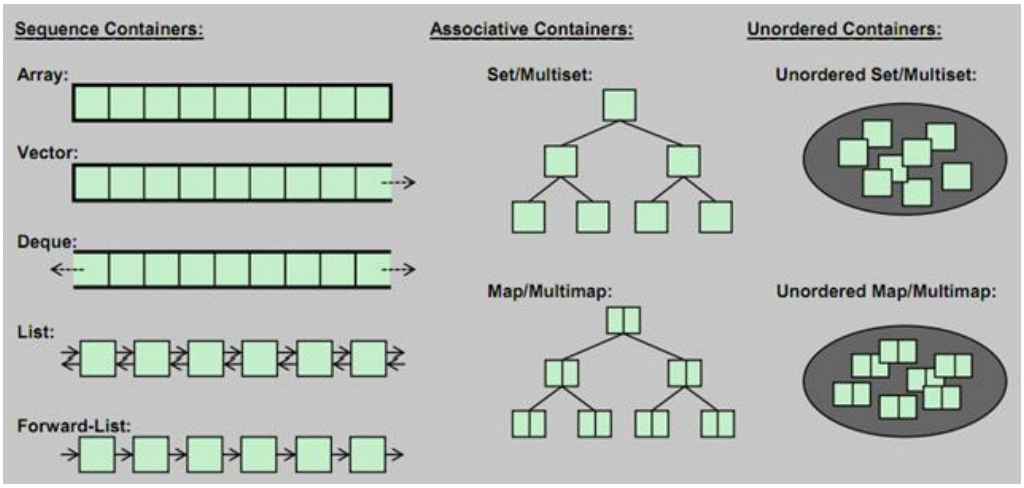
```
#include <iostream>
#include <vector>
using namespace std;

class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<"from"<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<"from"<<&other<<endl;
    }
    ~A()
```

```
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};

int main()
{
    vector<A*>  va;
    va.push_back(new A);
    va.push_back(new A);
    va.push_back(new A);
}
```

3.3.分类 category

3.3.1. 容器内存结构

STL 容器的分类与内存结构相关的。内存结构即分类标准,下图展示了分类与内存结构模型。



3.3.2. 分类

从上图示中，可以看出，总共分为三大类，一类是 Sequence Containers，即序列容器，一类是 Associative Containers，即关联容器，还有一类称为无序容器，通常将其划分到关联容器中。

还有一类没有图中体现出来，那就是 Adapt Containers 适配器容器。适配器容器，没在图中体现出来，因为没有独立的内存模型，是适配的其它已有的内存模型。

3.4.接口统一 Common Api

STL 尽最大程度作到了接口的统一，这样会带来更多的便利性和见名知义性。

3.5.容器适用场景 SWOT

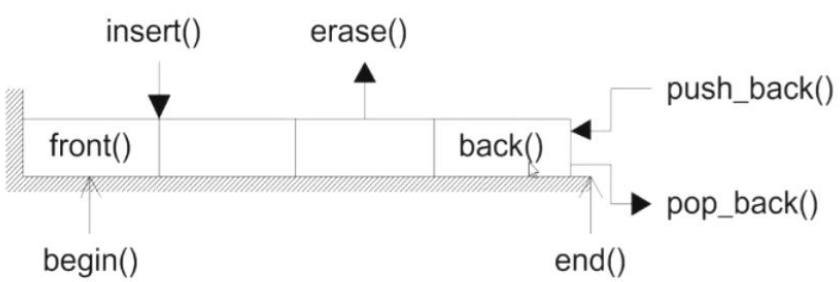
	Vector	Deque	List	Set	MultiSet	Map	MultiMap
内部结构	dynamic array	array of arrays	double linked list	binary tree	binary tree	binary tree	binary tree
随机存取	Yes	Yes	No	No	No	Yes(key)	No
搜索速度	慢	慢	很慢	快	快	快	快
快速插入移除	尾部	首尾	任何位置	--	--	--	--

4. 序列容器 Sequense

4.1.Vector 标准入门

4.1.1. Digram

4.1.1.1. 内存结构



向量（vector）

4.1.1.2. 声明与头文件

```
#include <vector>
namespace std {
template <typename T,
        typename Allocator = allocator<T> >
class vector;
}
```

4.1.1.3. 图示解析

4.1.2. Constructors and Destructor

4.1.2.1. api

Operation	Effect
vector<E> c	Default constructor; creates an empty vector without anyelements
vector<E> c(c2)	Copy constructor; creates a new vector as a copy of c2 (allelements are copied)
vector<E> c = c2	Copy constructor; creates a new vector as a copy of c2 (allelements are copied)
vector<E> c(rv)	Move constructor; creates a new vector, taking the contents of the rvalue rv (since C++11)
vector<E> c = rv	Move constructor; creates a new vector, taking the contents of the rvalue rv (since C++11)
vector<E> c(n)	Creates a vector with n elements created by the defaultconstructor
vector<E> c(n,elem)	Creates a vector initialized with n copies of element elem

vector<E> c(beg,end)	Creates a vector initialized with the elements of the range [beg,end)
vector<E> c(initlist)	Creates a vector initialized with the elements of initializer list initlist (since C++11)
vector<E> c = initlist	Creates a vector initialized with the elements of initializer list initlist (since C++11)
c.~vector()	Destroys all elements and frees the memory

4.1.2.2. test

4.1.3. Nonmodifying Operations

4.1.3.1. api

Operation	Effect
c.empty()	Returns whether the container is empty (equivalent to size()==0 but might be faster)
c.size()	Returns the current number of elements
c.max_size()	Returns the maximum number of elements possible
c.capacity()	Returns the maximum possible number of elements without reallocation
c.reserve(num)	Enlarges capacity, if not enough yet
c.shrink_to_fit()	Request to reduce capacity to fit number of elements (since C++11)
c1 == c2	Returns whether c1 is equal to c2 (calls == for the elements)
c1 != c2	Returns whether c1 is not equal to c2 (equivalent to !(c1==c2))
c1 < c2	Returns whether c1 is less than c2
c1 > c2	Returns whether c1 is greater than c2 (equivalent to c2<c1)
c1 <= c2	Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1))
c1 >= c2	Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2))

4.1.3.2. test

4.1.4. Assignments

4.1.4.1. api

Operation	Effect
c = c2	Assigns all elements of c2 to c
c = rv	Move assigns all elements of the rvalue rv to c (since C++11)
c = initlist	Assigns all elements of the initializer list initlist to c (since C++11)
c.assign(n,elem)	Assigns n copies of element elem
c.assign(beg,end)	Assigns the elements of the range [beg,end)

c.assign(initlist)	Assigns all the elements of the initializer list initlist
c1.swap(c2)	Swaps the data of c1 and c2
swap(c1,c2)	Swaps the data of c1 and c2

4.1.4.2. test

4.1.5. Element Access

4.1.5.1. api

Operation	Effect
c[idx]	Returns the element with index idx (no range checking)
c.at(idx)	Returns the element with index idx (throws range-error exception if idx is out of range)
c.front()	Returns the first element (no check whether a first element exists)
c.back()	Returns the last element (no check whether a last element exists)

4.1.5.2. test

4.1.6. Iterator Operations

4.1.6.1. api

Operation	Effect
c.begin()	Returns a random-access iterator for the first element
c.end()	Returns a random-access iterator for the position after the last element
c.cbegin()	Returns a constant random-access iterator for the first element (since C++11)
c.cend()	Returns a constant random-access iterator for the position after the last element (since C++11)
c.rbegin()	Returns a reverse iterator for the first element of a reverse iteration
c.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration
c.crbegin()	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
c.crend()	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

4.1.6.2. test

需要配合相应的迭代器使用。

```
int main()
{
    vector<int>::iterator;
    vector<int>::reverse_iterator;
    vector<int>::const_iterator;
    vector<int>::const_reverse_iterator;
```

```
    return 0;
}
```

4.1.7. Insert and Remove Operations

4.1.7.1. api

Operation	Effect
c.push_back(elem)	Appends a copy of elem at the end
c.pop_back()	Removes the last element (does not return it)
c.insert(pos,elem)	Inserts a copy of elem before iterator position pos and returns the position of the new element
c.insert(pos,n,elem)	Inserts n copies of elem before iterator position pos and returns the position of the first new element (or pos if there is no new element)
c.insert(pos,beg,end)	Inserts a copy of all elements of the range [beg,end) before iterator position pos and returns the position of the first new element (or pos if there is no new element)
c.insert(pos,initlist)	Inserts a copy of all elements of the initializer list initlist before iterator position pos and returns the position of the first new element (or pos if there is no new element; since C++11)
c.emplace(pos,args...)	Inserts a copy of an element initialized with args before iterator position pos and returns the position of the new element (since C++11)
c.emplace_back(args...)	Appends a copy of an element initialized with args at the end (returns nothing; since C++11)
c.erase(positr)	Removes the element at iterator position pos and returns the position of the next element
c.erase(begitr,enditr)	Removes all elements of the range [beg,end) and returns the position of the next element
c.resize(num)	Changes the number of elements to num (if size() grows new elements are created by their default constructor)
c.resize(num,elem)	Changes the number of elements to num (if size() grows new elements are copies of elem)
c.clear()	Removes all elements (empties the container)

4.1.7.2. test

```
int main()
{

    vector<Elem> coll;
    ...
    // remove first element with value val
```

```
    vector<Elem>::iterator pos;
    pos = find(coll.begin(),coll.end(),val);
    if (pos != coll.end()) {
        coll.erase(pos);
    }

}
```

4.1.8. 综合 Comprehensive Test

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{

    vector<string> sentence;

    sentence.reserve(5); //capacity
    sentence.push_back(string("Hello,"));
    sentence.insert(sentence.end(),{"how","are","you","?"});

    copy (sentence.cbegin(), sentence.cend(),
          ostream_iterator<string>(cout," "));
    cout << endl;

    cout << " max_size(): " << sentence.max_size() << endl;
    cout << " size(): " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    swap (sentence[1], sentence[3]);

    sentence.insert (
              find(sentence.begin(),sentence.end(),"?"),
              "always"
              );

    sentence.back() = "!";

//    "Hello,"  "you","are","how"  !

    copy (sentence.cbegin(), sentence.cend(),
          ostream_iterator<string>(cout," "));
    cout << endl;
```

```
    cout << " size(): " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    sentence.pop_back();
    sentence.pop_back();

    sentence.shrink_to_fit();

    cout << " size(): " << sentence.size() << endl;
    cout << " capacity(): " << sentence.capacity() << endl;

    return 0;
}
```

4.2.Vector 高级主题

4.2.1. 内存管理策略

4.2.1.1. size/capacity/resize/reserve

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> vi;
//    vi.reserve(16);
    for(int i=0; i<10; i++)
    {
        vi.push_back(i);
        cout<<"size: "<<vi.size()<<
             "capacity:"<<vi.capacity()<<endl;
    }

    vi.resize(1);
    cout<<"size:"<<vi.size()<<endl;
    cout<<"capacity:"<<vi.capacity()<<endl;

    vi.reserve(10);
    cout<<"size:"<<vi.size()<<endl;
    cout<<"capacity:"<<vi.capacity()<<endl;

    return 0;
}
```

4.2.1.2. Programing Tip

- 事先预估内存容量，避免内存的重复申请。
- resize 可以改变 size 的大小，变大等价于调用 push_back(),变小等于 pop_back() ，resize 不可以改变 capacity 的大小。
- reserve 只可变大 vector 的 capacity 而不可以使其减小。

4.2.2. 压入对象

4.2.2.1. class A

```
#include <iostream>
#include <vector>
using namespace std;


class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    vector<A> va(10);  //va;

    va.assign(3,A());
```

```
    vector<A>  va2;
    va2 = va;

    return 0;
}
```

4.2.2.2. Programing Tip

- 当向容器中压的是类对象成员时，最好要保证无参构造器的存在，类似于数组中的类对象一样。
- 在必要的情况下，要自实现拷贝构造和拷贝赋值。
- 压入栈中的对象元素在堆中，所申请的内存已托管，无需再次打理。

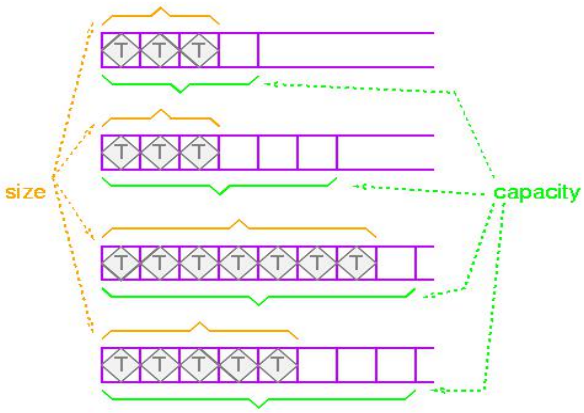
4.2.3. 空间与构造空间

4.2.3.1. resize&&reserve

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
```

```
};
int main()
{
   vector<A> va;
//   va.resize(10);
//   cout<<"size:"<<va.size()<<endl;
//   cout<<"capcity:"<<va.capacity()<<endl;

   va.reserve(20);
   cout<<"size:"<<va.size()<<endl;
   cout<<"capcity:"<<va.capacity()<<endl;

   return 0;
}
```

4.2.3.2. 图示 size 与 capcity



4.2.3.3. Programing Tip

- resize 会调用构造器，而 reserve 则不会调用构造器。
- 事先 reserve 出空间来，可以大量的减少不必要的内存拷贝。

4.2.4. 压入对象指针

4.2.4.1. pushback(A*)->erase(find(A*))

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    A *p1 = new A;
    A *p2 = new A;
    A *p3 = new A;

    vector<A *> vap;
    vap.push_back(p1);
    vap.push_back(p2);
    vap.push_back(p3);

    vector<A*>::iterator itr = find(vap.begin(),vap.end(),p1);
    if(itr != vap.end())
    {
//        delete *itr;
        vap.erase(itr);
    }

    cout<<"此上操作未导致申请的内存释放"<<endl;

    for(auto & itr:vap)
```

```
        delete itr;


    return 0;
}
```

4.2.4.2. Programing Tip

- 容器有内存托管的功能，但仅限于，被压入的实体元素。
- 若元素为指针类型，则指针被托管。而不是指针所指向的空间被托管。

4.2.5. 基本类型到对象排序

4.2.5.1. 基本数据类型

```
int main()
{
    vector<int> vi= {1,3,5,2,4,6};
    sort(vi.begin(),vi.end(),less<int>());
    for(auto &i:vi)
        cout<<i<<endl;
    return 0;

}
```

4.2.5.2. 自定义数据类型

对于自定义数据类型，要想实现排序，至少要自实现 operator<的自实现。less 就是调用了这个函数。

```
    friend bool operator<(const A & one,const A& another)
    {
        if(one._data <  another._data)
            return true;
        else
            return false;
    }

    bool operator<(const A &another)  const
    {
        if(this->_data <  another._data)
            return true;
        else
            return false;
    }
```

operator<可以实现为成员函数，也可以实现为友元，实现为成员函数时，别忘了将函数 const 化。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    friend bool operator<(const A & one,const A& another)
    {
        if(one._data <  another._data)
            return true;
        else
            return false;
    }

    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
        this->_data = other._data;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
        if(this == &other)
            return *this;
        this->_data = other._data;
    }

    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }

    int data()
    {
```

```
        return _data;
    }

private:
    int _data;
};
int main()
{
    vector<A> vi = {A(1),A(3),A(5),A(2),A(4),A(6)};

    sort(vi.begin(),vi.end(),not2(less<A>()));

    //not1 是构造一个与谓词结果相反的一元函数对象,
    //not2 是构造一个与谓词结果相反的二元函数对象。

    for(auto &obj:vi)
        cout<<obj.data()<<endl;

    return 0;

}
```

4.2.6. 高效的插入与删除 effective erase/insert

4.2.6.1. erase/pop_back&&insert/push_back

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
    }
```

```
    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};
int main()
{
    vector<A> va;
    va.reserve(100);
    for(int i=0; i<10; i++)
    {
        va.push_back(A());
    }
//    va.insert(va.begin(),100);//在头部插入会带来大量的拷贝

    //如下的插入才是高效的
//    va.push_back(A());
//    va.insert(va.end(),A());

//    va.erase(va.begin());//在头部删除会带来大量的拷贝

    //如下的删除才是高效的
    va.pop_back();
    va.erase(va.end()-1);

    return 0;
}
```

4.2.6.2. Programing Tip

- 尾部的插入和删除是高效的 O(1),其它位置则是低效的 O(n)
- 故对于 vector 而言，提倡使用 push_back()和 pop_back()，仅量少用 insert 和 erase

4.2.7. erase+find+remove

4.2.7.1. find remove 基本元素

erase 后面只能跟迭代器，而 find 可以返回找到元素的迭代器，remove 可以高效的移动覆盖要删除的元素。

```
int main()
{
```

```
    vector<int> vi = {1,3,5,7,9,2,4,6,8,10};
//    vi.erase(vi.begin());
//    vi.erase(vi.begin()+1,vi.end()-1);



//    auto itrfind  = find(vi.begin(),vi.end(),1000);
//    if(itrfind != vi.end())
//        vi.erase(itrfind);
//    else {
//       cout<<"find none"<<endl;
//    }

//    for (auto & i: vi) {
//        cout<<i<<endl;
//    }
    auto itr = std::remove(vi.begin(),vi.end(),9);
//    if(itr != vi.end())
//    {
//        vi.pop_back();

//    }
//    for (auto & i: vi) {
//        cout<<i<<endl;
//    }
    vi.erase(itr,vi.end());
    for (auto & i: vi) {
        cout<<i<<endl;
    }

    return 0;
}
```

4.2.7.2. find remove 对象成员

若是对象成员，sort 到少需要提供 operator<重载，而 find 和 remove 则，至少需要提供 operator==重载。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;


class  A
{
public:
    A(int i):_i(i){
```

```
    }
    bool operator<( const A & another) const
    {
        return this->_i > another._i? true :false;
    }

    bool operator==( const A & another)
    {
        return this->_i  == another._i? true :false;
    }

    void dis()
    {
        cout<<_i<<endl;
    }
protected:
    int _i;
};

int main()
{
    vector<A> va;
    va.push_back(A(1));
    va.push_back(A(3));
    va.push_back(A(2));
    va.push_back(A(5));
    va.push_back(A(7));
//    sort(va.begin(),va.end(),not2(less<A>()));
//    for( auto & item: va)
//        item.dis();

    auto itr = find(va.begin(),va.end(),A(5));
//    itr->dis();
    va.erase(itr);
    for( auto & item: va)
        item.dis();

}
```

4.2.8. 失效的迭代器 invalid iterator

4.2.8.1. 问题由来

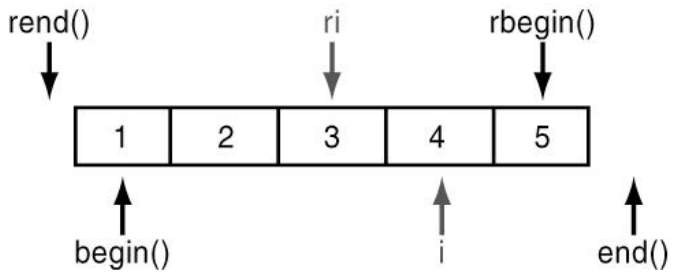
```
#include <iostream>
#include <vector>
using namespace std;
```

```
int main()
{
    vector<int> vi;
    int data[10] = {1,3,5,7,9,2,4,6,8,10};
    vi.assign(data,data+10);
    vector<int>::iterator itr = vi.begin();

    for(;itr != vi.end(); ++itr)
    {
        if(*itr%2==0)
            vi.erase(itr);
        else
        {
            cout<<*itr<<endl;
        }
    }
    return 0;
}
```

4.2.8.2. 失效原因

vector 是一个顺序容器，在内存中是一块连续的内存，当删除一个元素后，内存中的数据会发生移动，以保证数据的紧凑。所以删除一个数据后，其他数据的地址发生了变化，之前获取的迭代器根据原有的信息就访问不到正确的数据。

插入一个元素，会导致迭代器失效吗？比如在奇数元素的位置插入 100?



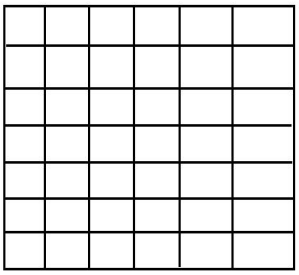
4.2.8.3. 解决方案

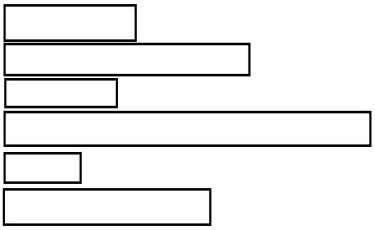
```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    vector<int> vi;
    int data[10] = {1,3,5,7,9,2,4,6,8,10};
    vi.assign(data,data+10);
    vector<int>::iterator itr = vi.begin();
```

```
    for(;itr != vi.end();)
    {
        if(*itr%2==0)
            itr = vi.erase(itr);
        else
        {
            cout<<*itr<<endl;
             ++itr;
        }
    }
    return 0;
}
```

4.2.9. 二维及多维空间生成

4.2.9.1. 二维空间





4.2.9.2. vector 嵌套

如何能过 vector 生成二维及多维空间呢？答案是 vector 中嵌套 vector，注意嵌套的语法格式。

```
#include <iostream>
#include <vector>
using namespace std;


int main()
{
    vector<vector<int> > arr;
    arr.resize(5);
    for(int i=0; i<5; i++)
    {
        arr[i].resize(5);
    }

//    vector<int> vi(5);
```

```
//    for(int i=0; i<5; i++)
//    {
//       arr.push_back(vi);
//    }

    for(int i=0; i<5; i++)
    {
       for(int j=0; j<5; j++)
       {
          cout<<arr[i][j]<<" ";
       }
       cout<<endl;
    }
    return 0;
}

//vector<vector<int> > arr(5,vector<int>(5,0)); 这样写可以吗?
```

分析一下，如下代码的语义：

```
#ifndef FIELD_H
#define FIELD_H
#include <vector>
#include <iostream>
using namespace std;


namespace SweepMines{
namespace Model{

typedef vector<int>       CellColumn;
typedef vector<CellColumn>  CellMatrix;

class Field
{
public:
    Field();
    ~Field();
    int getWidth() const {return width;}
    int getHeight() const {return height;}
    int getMines() const {return mines;}
    const CellMatrix & getCells(){return cells;}
    void deployMines();
    void dump();

    void customize(int w, int h ,int m);

protected:
    void initCells();
```

```
    void updateSurrounding(int x, int y);

    int width;
    int height;
    int mines;

/*
  cell[x][y] < 0 has a mine in the cell
  cell[x][y] >= 0  the cout mines in the neighborhood
  */
    CellMatrix cells;
};
```

```
#include "field.h"
#include <iostream>
#include <time.h>
#include <string.h>
#include <stdlib.h>
#include <QDebug>

namespace SweepMines{
namespace Model{

Field::Field():
    width(5),
    height(3),
    mines(3)
{
    initCells();
    deployMines();
}

Field::~Field()
{
    cells.clear();
}

void Field::initCells()
{
    for(int x=0; x<width; x++)
    {
        cells.push_back(CellColumn(height));
    }
}
```

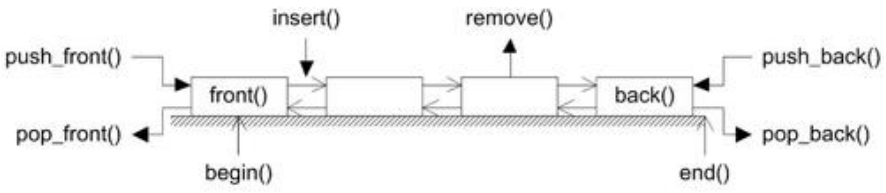
```
void Field::deployMines()
{
//初始化 stl 空间
    for(int x=0; x<width; x++)
        for(int y=0; y<height; y++)
            cells[x][y] = 0;

    qDebug()<<width<<" "<<height<<"  "<<mines;
    int x,y;
    int cpmines = mines;
    srand(time(NULL));
    while(cpmines)
    {
//       srand(time(NULL));xiao lu xiao lu
        x = rand()%width;
        y = rand()%height;
        if(cells[x][y]>=0)
        {
            cells[x][y] = -1;
            cpmines--;
            updateSurrounding(x-1,y-1);
            updateSurrounding(x-1,y);
            updateSurrounding(x-1,y+1);
            updateSurrounding(x,y+1);
            updateSurrounding(x+1,y+1);
            updateSurrounding(x+1,y);
            updateSurrounding(x+1,y-1);
            updateSurrounding(x,y-1);
        }
    }
    dump();
}

void Field::updateSurrounding(int x, int y)
{
    if(x>=0&&x<width&&y>=0&&y<height)
    {
        if(cells[x][y] != -1)
            cells[x][y]++;
    }
}

void Field::dump()
{
    for(int y=0; y<height; y++)
    {
        for(int x=0; x<width; x++)
        {
```

```
            std::cout<<cells[x][y]<<'\t';
        }
        std::cout<<std::endl;
    }
}
void Field::customize(int w, int h ,int m)
{
    width = w;
    height = h;
    mines = m;

    cells.clear();

    initCells();
    deployMines();
}
}
}
```

4.3.List 标准入门

4.3.1. Digram

4.3.1.1. 内存结构



列表（list）

4.3.1.2. 类的声明与头文件

```
#include <list>
namespace std {
template <typename T,
        typename Allocator = allocator<T> >
class list;
}
```

4.3.1.3. 图示解析

4.3.2. Constructors and Destructor

4.3.2.1. api

Operation	Effect
list<Elem> c	Default constructor; creates an empty list without any elements
list<Elem> c(c2)	Copy constructor; creates a new list as a copy of c2 (all elements are copied)
list<Elem> c = c2	Copy constructor; creates a new list as a copy of c2 (all elements are copied)
list<Elem> c(rv)	Move constructor; creates a new list, taking the contents of the rvalue rv (since C++11)
list<Elem> c = rv	Move constructor; creates a new list, taking the contents of the rvalue rv (since C++11)
list<Elem> c(n)	Creates a list with n elements created by the default constructor
list<Elem> c(n,elem)	Creates a list initialized with n copies of element elem
list<Elem> c(beg,end)	Creates a list initialized with the elements of the range [beg,end)
list<Elem> c(initlist)	Creates a list initialized with the elements of initializer list initlist (since C++11)
list<Elem> c = initlist	Creates a list initialized with the elements of initializer list initlist (since C++11)
c.~list()	Destroys all elements and frees the memory

4.3.2.2. test

4.3.3. Nonmodifying Operations

4.3.3.1. api

Operation	Effect
c.empty()	Returns whether the container is empty (equivalent to size()==0 but might be faster)
c.size()	Returns the current number of elements
c.max_size()	Returns the maximum number of elements possible
c1 == c2	Returns whether c1 is equal to c2 (calls == for the elements)
c1 != c2	Returns whether c1 is not equal to c2 (equivalent to !(c1==c2))
c1 < c2	Returns whether c1 is less than c2
c1 > c2	Returns whether c1 is greater than c2 (equivalent to c2<c1)
c1 <= c2	Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1))
c1 >= c2	Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2))

4.3.3.2. test

4.3.4. Assignment Operations

4.3.4.1. api

Operation	Effect
c = c2	Assigns all elements of c2 to c
c = rv	Move assigns all elements of the rvalue rv to c (since C++11)
c = initlist	Assigns all elements of the initializer list initlist to c (since C++11)
c.assign(n,elem)	Assigns n copies of element elem
c.assign(beg,end)	Assigns the elements of the range [beg,end)
c.assign(initlist)	Assigns all the elements of the initializer list initlist
c1.swap(c2)	Swaps the data of c1 and c2
swap(c1,c2)	Swaps the data of c1 and c2

4.3.4.2. test

4.3.5. Element Access

4.3.5.1. api

Operation	Effect
c.front()	Returns the first element (no check whether a first element exists)
c.back()	Returns the last element (no check whether a last element exists)

4.3.5.2. test

4.3.6. Iterator Operations

4.3.6.1. api

Operation	Effect
c.begin()	Returns a bidirectional iterator for the first element
c.end()	Returns a bidirectional iterator for the position after the last element
c.cbegin()	Returns a constant bidirectional iterator for the first element (since C++11)
c.cend()	Returns a constant bidirectional iterator for the position after the last element (since C++11)
c.rbegin()	Returns a reverse iterator for the first element of a reverse iteration
c.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration
c.crbegin()	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
c.crend()	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

4.3.6.2. test

4.3.7. Inserting and Removing Elements

4.3.7.1. api

Operation	Effect
c.push_back(elem)	Appends a copy of elem at the end
c.pop_back()	Removes the last element (does not return it)
c.push_front(elem)	Inserts a copy of elem at the beginning
c.pop_front()	Removes the first element (does not return it)
c.insert(pos,elem)	Inserts a copy of elem before iterator position pos and returns the position of the new element
c.insert(pos,n,elem)	Inserts n copies of elem before iterator position pos and returns the position of the first new element (or pos if there is no new element)
c.insert(pos,beg,end)	Inserts a copy of all elements of the range [beg,end) before iterator position pos and returns the position of the first new element (or pos if there is no new element)
c.insert(pos,initlist)	Inserts a copy of all elements of the initializer list initlist before iterator position pos and returns the position of the first new element (or pos if there is no new element; since C++11)
c.emplace(pos,args...)	Inserts a copy of an element initialized with args before iterator position pos and returns the position of the new element (since C++11)
c.emplace_back(args...)	Appends a copy of an element initialized with args at the end (returns nothing; since C++11)
c.emplace_front(args...)	Inserts a copy of an element initialized with args at the beginning (returns nothing; since C++11)
c.erase(pos)	Removes the element at iterator position pos and returns the position of the next element
c.erase(beg,end)	Removes all elements of the range [beg,end) and returns the position of the next element
c.remove(val)	**Removes all elements with value val**
c.remove_if(op)	**Removes all elements for which op(elem) yields true**
c.resize(num)	Changes the number of elements to num (if size() grows new elements are created by their default constructor)
c.resize(num,elem)	Changes the number of elements to num (if size() grows new elements are copies of elem)
c.clear()	Removes all elements (empties the container)

4.3.7.2. erase_test

```
#include <list>
#include <iostream>
#include <iterator>
```

```
using namespace std;

int main( )
{
    list<int> c={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    for (auto &i : c) {
        cout << i << " ";
    }
    cout << '\n';

    c.erase(c.begin());

    for (auto &i : c) {
        cout << i << " ";
    }
    cout << '\n';

    list<int>::iterator range_begin = c.begin();
    list<int>::iterator range_end = c.begin();
    advance(range_begin,2);
    advance(range_end,5);

    c.erase(range_begin, range_end);

    for (auto &i : c) {
        cout << i << " ";
    }
    cout << '\n';
}
```

4.3.7.3. remove_test

```
#include <list>
#include <iostream>
using namespace std;

int main()
{
    list<int> l = { 1,100,2,3,10,1,11,-1,12 };
    l.remove(1);


    l.remove_if([](int n){ return n > 10; });
    for (int n : l) {
        cout << n << ' ';
    }
    cout << '\n';
}
```

4.3.8. Special Modifying Operations

4.3.8.1. api

Operation	Effect
c.unique()	Removes duplicates of consecutive elements with the same value
c.unique(op)	Removes duplicates of consecutive elements, for which op() yields true
c.splice(pos,c2)	Moves all elements of c2 to c in front of the iteratorposition pos
c.splice(pos,c2,c2pos)	Moves one element at c2pos in c2 in front of pos of list c (c and c2 may be identical)
c.splice (pos,c2, c2beg,c2end)	Moves all elements of the range [c2beg,c2end) in c2 in front of pos of list c (c and c2 may be identical)
c.sort()	Sorts all elements with operator <
c.sort(op)	Sorts all elements with op()
c.merge(c2)	Assuming that both containers contain the elements sorted, moves all elements of c2 into c so that all elements are merged and still sorted
c.merge(c2,op)	Assuming that both containers contain the elements sorted due to the sorting criterion op(), moves all elements of c2 into c so that all elements are merged and still sorted according to op()
c.reverse()	Reverses the order of all elements

list 中的自带的 sort,因为 list 的迭代器是双向迭代器,所有不能使用 STL 算法中的 sort（随机访问迭代器才能）。

库函数 int string 类型中已经包含< 符号，这时候默认的是升序排列。当需要降序时，需要自定义 compare 函数，实现即 sort(compare)。

当 list 的类型是结构体类型或者类类型时，需要自己写重载操作符<。

4.3.8.2. uniqe_test

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> x = {1, 2, 2, 3, 3, 2, 1, 1, 2};
    cout << "contents before:";
    for (auto val : x)
        cout << ' ' << val;
    cout << '\n';
    x.unique();
    cout << "contents after unique():";
    for (auto val : x)
        cout << ' ' << val;
    cout << '\n';
    return 0;
```

```
}
```

4.3.8.3. splice_test

Transfers elements from one list to another. No elements are copied or moved, only the internal pointers of the list nodes are re-pointed.

```
#include <iostream>
#include <list>
using namespace std;

ostream& operator<<(ostream& ostr,
                const list<int>& list)
{
    for (auto &i : list) {
       ostr << " " << i;
    }
    return ostr;
}

int main ()
{
    list<int> list1 = { 1, 2, 3, 4, 5 };
    list<int> list2 = { 10, 20, 30, 40, 50 };

    auto it = list1.begin();
    advance(it, 2);

    list1.splice(it, list2);

    cout << "list1: " << list1 << "\n";
    cout << "list2: " << list2 << "\n";

    list2.splice(list2.begin(), list1, it, list1.end());

    cout << "list1: " << list1 << "\n";
    cout << "list2: " << list2 << "\n";

    return 0;
}
```

4.3.8.4. merge_test

Merges two sorted lists into one. The lists should be sorted into ascending order.

```
#include <iostream>
#include <list>
using namespace std;

ostream& operator<<(ostream& ostr,
                const list<int>& list)
```

```
{
    for (auto &i : list) {
        ostr << " " << i;
    }
    return ostr;
}

int main()
{
    list<int> list1 = { 5,9,0,1,3 };
    list<int> list2 = { 8,7,2,6,4 };

    list1.sort();
    list2.sort();
    cout << "list1:  " << list1 << "\n";
    cout << "list2:  " << list2 << "\n";
    list1.merge(list2);
    cout << "list1: " << list1 << "\n";
    cout << "list2: " << list2 << "\n";
    return 0;
}
```

4.3.9. Comprehensive Test

```
#include <list>
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;
void printLists (const list<int>& l1,
                 const list<int>& l2)
{
    cout << "list1: ";
    copy (l1.cbegin(), l1.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl << "list2: ";
    copy (l2.cbegin(), l2.cend(),
          ostream_iterator<int>(cout," "));
    cout << endl << endl;
}
int main()
{

    list<int> list1, list2;

    for (int i=0; i<6; ++i)
    {
        list1.push_back(i);
```

```
        list2.push_front(i);
    }
    printLists(list1, list2);

    list2.splice(find(list2.begin(),list2.end(),3),list1);

    printLists(list1, list2);

    list2.splice(list2.end(),
                 list2,
                 list2.begin());

    list2.sort();
    list1 = list2;

    list2.unique();
    printLists(list1, list2);

    list1.merge(list2);
    printLists(list1, list2);

    return 0;
}
```

4.4.List 的高级主题

4.4.1. 随机访问 random access

4.4.1.1. front/back

```
#include <iostream>
#include <list>

using namespace std;

int main()
{
    list<int>  li = {1,3,5,7,9,2,4,6,8,10};

    cout<<"front:"<<li.front()<<endl;
    cout<<"end  :"<<li.back()<<endl;

//    cout<<"front:"<<li[0]<<endl;

    return 0;
}
```

4.4.1.2. Programing Tip

- 不提供下标访问([], at)，随机访问是低效的,第一和最后一个元素访问是高效的。

4.4.2. list 的内部函数(list::sort)

list 容器由于采用了双向迭代器，不支持随机访问，所以标准库的 merge(), sort()等功能函数都不适用，list 单独实现了 merge()，sort()等函数。

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
    list<int>  li = {1,3,5,7,9,2,4,6,8,10};

    li.sort();
    for(auto &i:li)
        cout<<i<<endl;

    list<int> li2 = {11,12,15,13,14};

    li.merge(li2);
    li.sort();
    for(auto &i:li)
        cout<<i<<"\t";
    cout<<endl;

    for(auto &i:li2)
        cout<<i<<"\t";
    cout<<endl;

    return 0;
}
```

4.4.3. 高效的插入与删除 effective insert erase

4.4.3.1. insert /erase

```
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;
```

```
class A
{
public:
    A()
    {
        cout<<"无参构造函数"<<this<<endl;
    }
    A(int i):_data(i)
    {
        cout<<"有参构造函数"<<this<<endl;
    }
    A(const A & other)
    {
        cout<<"拷贝构造"<<this<<" from "<<&other<<endl;
    }

    A& operator=(const A & other)
    {
        cout<<"拷贝赋值"<<this<<" from "<<&other<<endl;
    }
    ~A()
    {
        cout<<"析构函数"<<this<<endl;
    }
private:
    int _data;
};


int main()
{
    list<A> la;
    la.assign(10,A());

    la.erase(la.begin());
    la.insert(la.begin(),A());

    return 0;
}
```

4.4.3.2. Programing Tip

- 插入和删除是高效的，元素不需移动，内存仅是指针移动。

4.4.4. 失效的迭代器 invalid Iterator

4.4.4.1. valid or invalid

```
#include <iostream>
#include <list>
```

```
#include <iterator>
#include <algorithm>

using namespace std;

int main()
{
    list<int>  li = {1,3,5,7,9,2,4,6,8,10};

    for(auto itr = li.begin(); itr != li.end();)
    {
        if(*itr%2 == 0)
            itr = li.erase(itr);
        else
            cout<<*itr<<endl;
    }
    for(auto itr = li.begin(); itr != li.end();)
    {
        if(*itr%2 == 0)
            itr = li.erase(itr);
        else
          ++itr;
    }


    return 0;
}
```

4.4.4.2. remove/remove_if

4.4.4.3. Programing Tip

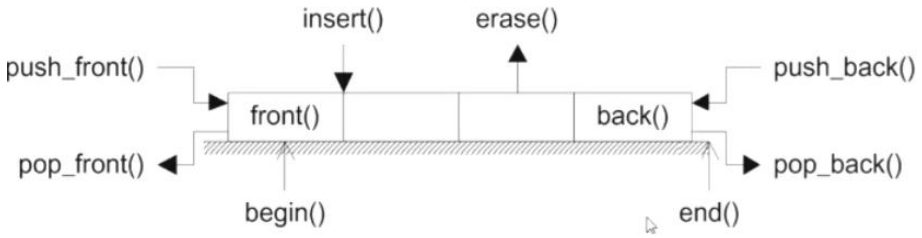
- 插入和删除不会使迭代器失效。

4.5.Dequeue 标准入门

Double end Queue，A deque (pronounced “deck”)。

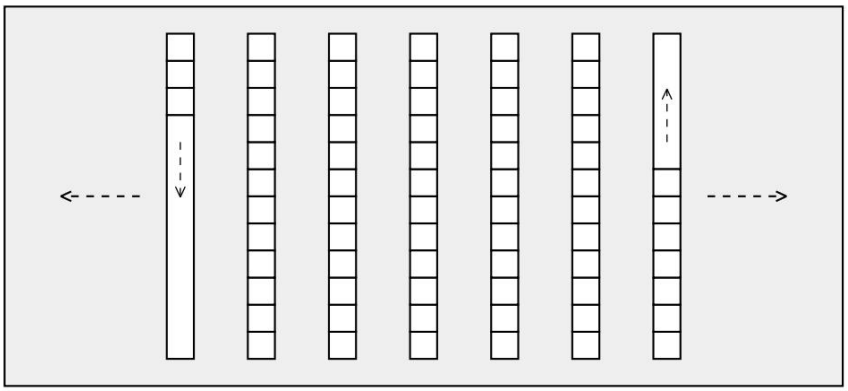
4.5.1. Digram

4.5.1.1. 逻辑结构



双端队列（deque）

4.5.1.2. 内存结构



4.5.1.3. 声明及头文件

```
#include <deque>
namespace std {
template <typename T,
        typename Allocator = allocator<T> >
class deque;
}
```

4.5.2. 栈操作

4.5.2.1. stack

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<int> di;
    cout<<di.size()<<endl;
    for(int i=0; i<10; i++)
    {
        di.push_back(i);
    }
    cout<<di.size()<<endl;
    while(!di.empty())
    {
        cout<<di.back()<<" ";
        di.pop_back();
    }
    cout<<endl<<di.size()<<endl;
    return 0;
}
```

4.5.3. 队列操作

4.5.3.1. queue

```
#include <iostream>
#include <deque>

using namespace std;

int main()
{
    deque<int> di;
```

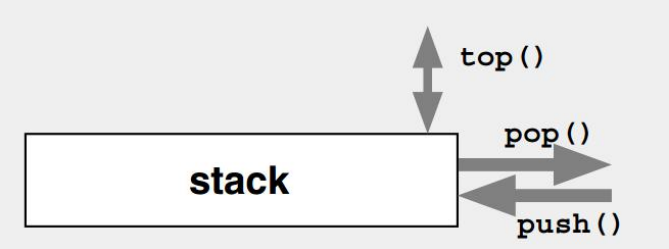
```
    cout<<di.size()<<endl;
    for(int i=0; i<10; i++)
    {
        di.push_back(i);
    }
    cout<<di.size()<<endl;
    while(!di.empty())
    {
        cout<<di.front()<<" ";
        di.pop_front();
    }
    cout<<endl<<di.size()<<endl;
    return 0;
}
```

5. 容器适配器 adapter

STL 提供了三个容器适配器：queue、priority_queue、stack。这些适配器都是包装了 vector、list、deque 中某个顺序容器的包装器。注意：适配器没有提供迭代器，也不能同时插入或删除多个元素。

5.1.栈 stack

5.1.1. diagram

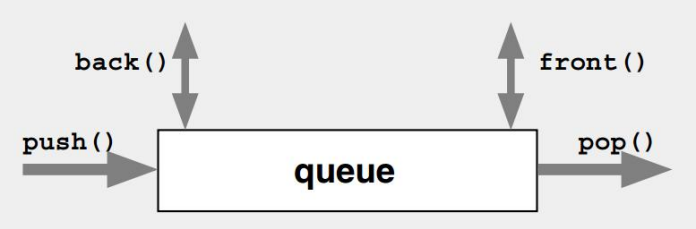


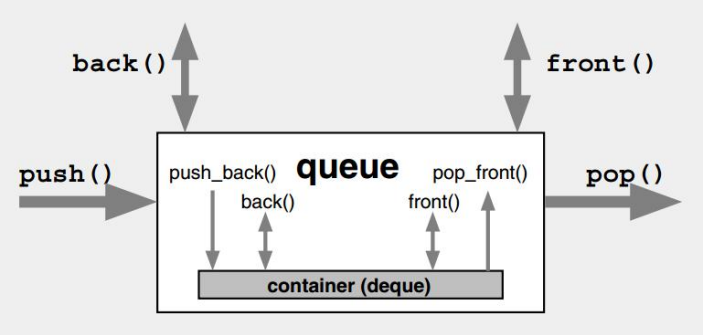
5.1.2. 应用

```
#include <iostream>
#include <stack>

using namespace std;

int main()
{
    stack<int> si;
    cout<<si.size()<<endl;

    for(int i=0; i<10; i++)
    {
        si.push(i);
    }
    cout<<si.size()<<endl;
    while(!si.empty())
    {
        cout<<si.top();
        si.pop();
    }
    cout<<endl<<si.size()<<endl;
    return 0;
}
```

5.2.队列 queue

5.2.1. diagram





5.2.2. 应用

```
#include <iostream>
#include <queue>

using namespace std;

int main()
{
    queue<int> qi;
    cout<<qi.size()<<endl;

    for(int i=0; i<10; i++)
    {
        qi.push(i);
    }

    while(!qi.empty())
    {
        cout<<qi.front()<<" ";
        qi.pop();
    }

    cout<<endl<<qi.size()<<endl;
```

```
    return 0;
}
```

5.3.优先队列 priority Queue

5.3.1. 定义

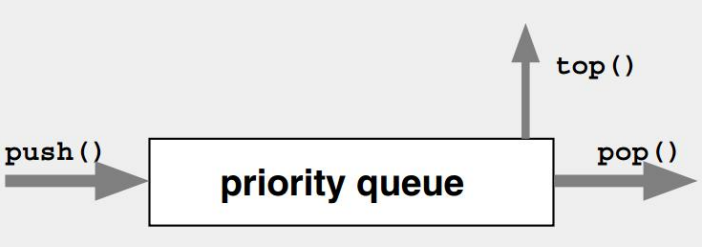
5.3.1.1. 优先队列

优先队列是队列的一种，不过它可以按照自定义的一种方式（数据的优先级）来对队列中的数据进行动态的排序

每次的 push 和 pop 操作，队列都会动态的调整，以达到我们预期的方式来存储。

例如：我们常用的操作就是对数据排序，优先队列默认的是数据大的优先级高所以我们无论按照什么顺序 push 一堆数，最终在队列里总是 top 出最大的元素。

5.3.1.2. 结构 diagram



5.3.1.3. 声明及头文件

```
namespace std {
template <typename T,
        typename Container = vector<T>,
        typename Compare = less<typename Container::value_type>>
class priority_queue;
}
```

T 为数据类型， Container 为保存数据的容器，Compare 为元素比较方式。如果不写后两个参数，那么容器默认用的是 vector，比较方式默认用 operator<，也就是优先队列是大顶堆，队头元素最大。

5.3.2. 应用

5.3.2.1. 构造规则

```
#include <queue>
#include <vector>
#include <iostream>
#include <functional>
```

```
using namespace std;

int main()
{
    priority_queue<int> c1;
    c1.push(5);
    cout << c1.size() << '\n';

    priority_queue<int> c2(c1);
    cout << c2.size() << '\n';

    vector<int> vec={3, 1, 4, 1, 5};
    priority_queue<int> c3(less<int>(), vec);
    cout << c3.size() << '\n';

    return 0;
}
```

5.3.2.2. 默认规则

可以支持，以 T 和 Container<T>和比较器(less<T>)进行实现化。比较器是可以支持自实的。

```
#include <functional>
#include <queue>
#include <vector>
#include <iostream>

using namespace std;

template<typename T>
void print_queue(T& q)
{
    while(!q.empty()) {
        cout << q.top() << " ";
        q.pop();
    }
    cout << '\n';
}

class compare
{
public:
    bool operator()(int x, int y)
    {
        return x > y;
    }
};
```

```
int main() {
    priority_queue<int> q;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q.push(n);

    print_queue(q);

    priority_queue<int, vector<int>, greater<int> > q2;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q2.push(n);

    print_queue(q2);


    priority_queue<int,vector<int>,compare> q3;

    for(int n : {1,8,5,6,3,4,0,9,7,2})
        q3.push(n);

    print_queue(q3);

}
```

5.3.2.3. 自定义规则

首先按优先级的大小入队，如果优先级相等，则按其它的优先级进行入队。可以支持多级优先级。

```
#include <queue>
#include <cstring>
#include <cstdio>
using namespace std;
//结构体
struct Node
{
    Node(int nri, char *pszName)
    {
        priority = nri;
        strcpy(szName, pszName);
    }
    char szName[20];
    int priority;
};
//结构体的比较方法 改写 operator()
class  NodeCmp
```

```
{
public:
    bool operator()(const Node &na, const Node &nb)
    {
        if (na.priority != nb.priority)
            return na.priority > nb.priority;
        else
            return strcmp(na.szName, nb.szName) < 0;
    }
};
void PrintfNode(const Node &na)
{
    printf("%s %d\n", na.szName, na.priority);
}
int main()
{
    //优先级队列默认是使用 vector 作容器，底层数据结构为堆。
    priority_queue<Node, vector<Node>, NodeCmp> a;
    //有 5 个人进入队列
    a.push(Node(5, "abc"));
    a.push(Node(3, "bac"));
    a.push(Node(1, "cba"));
    a.push(Node(5, "aaa"));
    //队头的 2 个人出队

    PrintfNode(a.top());
    a.pop();
    PrintfNode(a.top());
    a.pop();

    printf("--------------------\n");


    //再进入 3 个人
    a.push(Node(2, "bbb"));
    a.push(Node(2, "ccc"));
    a.push(Node(3, "aaa"));
    //所有人都依次出队
    while (!a.empty())
    {
        PrintfNode(a.top());
        a.pop();
    }

    //2 ccc 2 bbb 3 aaa  5abc 5aaa

    return 0;
}
```

6. 关联容器 Associative

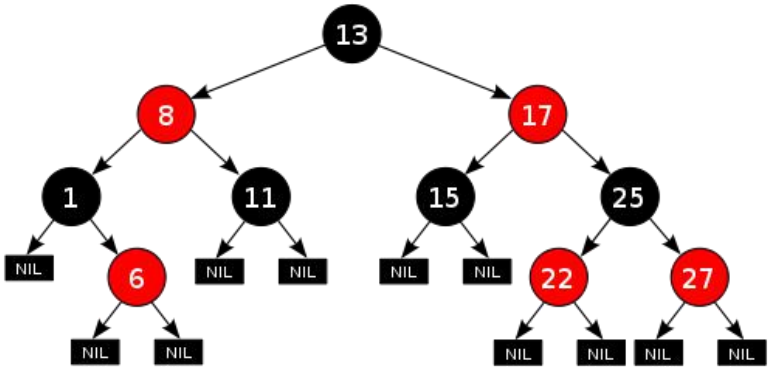
6.1.map

map 的特性是，所有元素会根据元素的键值自动被排序，map 的所有元素都是 pair，同时拥有实值(Value)和键值(Key)。

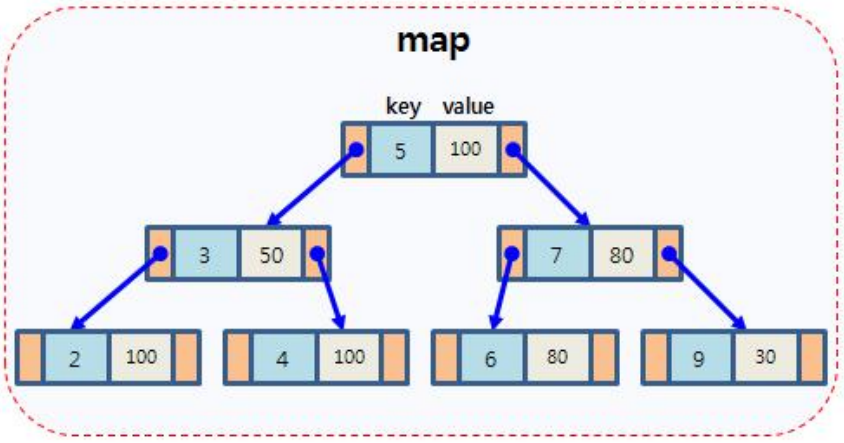
pair 的第一元素被视为键值，第二个元素被视为实值。map 不允许两个元素拥有相同的键值。

6.1.1. Diagram

6.1.1.1. 内存结构



6.1.1.2. pair



```
template<class _T1, class _T2>
struct pair{
    typedef _T1 first_type;
    typedef _T2 second_type;
    _T1 first;        // public
    _T2 second;       // public
    pair()
        : first(), second() { }
    pair(const _T1& __a, const _T2& __b)
        : first(__a), second(__b) { }
}
```

6.1.1.3. value_type

```
typedef pair<const Key, T> value_type;
```

6.1.1.4. 声明与头文件

```
#include <map>
namespace std {
template <typename Key,
          typename T,
          typename Compare = less<Key>,
          typename Allocator = allocator<pair<const Key,T> > >
class map;
template <typename Key,
          typename T,
          typename Compare = less<Key>,
          typename Allocator = allocator<pair<const Key,T> > >
class multimap;
}
```

6.1.2. Constructors and Destructor

6.1.2.1. api

Operation	Effect
map c	Default constructor; creates an empty map/multimap without any elements
map c(op)	Creates an empty map/multimap that uses op as the sorting criterion
map c(c2)	Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)
map c = c2	Copy constructor; creates a copy of another map/multimap of the same type (all elements are copied)

map c(rv)	Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue rv (since C++11)
map c = rv	Move constructor; creates a new map/multimap of the same type, taking the contents of the rvalue rv (since C++11)
map c(beg,end)	Creates a map/multimap initialized by the elements of the range [beg,end)
map c(beg,end,op)	Creates a map/multimap with the sorting criterion op initialized by the elements of the range [beg,end)
map c(initlist)	Creates a map/multimap initialized with the elements of initializer list initlist (since C++11)
map c = initlist	Creates a map/multimap initialized with the elements of initializer list initlist (since C++11)
c.~map()	Destroys all elements and frees the memory

上表中的 map 可为如下几种类型

Operation	Effect
map<Key,Val>	A map that by default sorts keys with less<> (operator <)
map<Key,Val,Op>	A map that by default sorts keys with Op
multimap<Key,Val>	A multimap that by default sorts keys with less<> (operator <)
multimap<Key,Val,Op>	A multimap that by default sorts keys with Op

6.1.2.2. test

```
#include <iostream>
#include <map>
#include <functional>
using namespace std;

int main()
{
    map<int,string > mis;
    map<int,string,less<int>> mis2;
    map<int,string > mis3(greater<int>());
    return 0;
}
```

6.1.3. Nonmodifying Operations

6.1.3.1. api

Operation	Effect
c.key_comp()	Returns the comparison criterion
c.value_comp()	Returns the comparison criterion for values as a whole (an object that compares the key in a key/value pair)
c.empty()	Returns whether the container is empty (equivalent to size()==0 but might be faster)
c.size()	Returns the current number of elements

c.max_size()	Returns the maximum number of elements possible
c1 == c2	Returns whether c1 is equal to c2 (calls == for the elements)
c1 != c2	Returns whether c1 is not equal to c2 (equivalent to !(c1==c2))
c1 < c2	Returns whether c1 is less than c2
c1 > c2	Returns whether c1 is greater than c2 (equivalent to c2<c1)
c1 <= c2	Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1))
c1 >= c2	Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2))

6.1.3.2. test

```
#include <iostream>
#include <map>
#include <functional>
using namespace std;

int main()
{
    map<int,string > mis;
    if(mis.empty())
    {
        cout<<"map empty"<<endl;
        cout<<"map size:"<<mis.size()<<endl;
    }

    mis[1] = "a";
    mis[2] = "ab";
    mis[3] = "abc";
    cout<<"map size:"<<mis.size()<<endl;
    cout<<"map max_size:"<<mis.max_size()<<endl;
    return 0;
}
```

6.1.4. Assignments

6.1.4.1. api

Operation	Effect
c = c2	Assigns all elements of c2 to c
c = rv	Move assigns all elements of the rvalue rv to c (since C++11)
c = initlist	Assigns all elements of the initializer list initlist to c (since C++11)
c1.swap(c2)	Swaps the data of c1 and c2
swap(c1,c2)	Swaps the data of c1 and c2

6.1.4.2. test

```
#include <iostream>
#include <map>
```

```
#include <functional>
using namespace std;

int main()
{
    map<int,string > mis;
    mis = {
        pair<int,string>(1,"a"),
        pair<int,string>(2,"abc")
    };
    return 0;
}
```

6.1.5. Iterator Functions

6.1.5.1. api

Operation	Effect
c.begin()	Returns a bidirectional iterator for the first element
c.end()	Returns a bidirectional iterator for the position after the last element
c.cbegin()	Returns a constant bidirectional iterator for the first element (since C++11)
c.cend()	Returns a constant bidirectional iterator for the position after the last element (since C++11)
c.rbegin()	Returns a reverse iterator for the first element of a reverse iteration
c.rend()	Returns a reverse iterator for the position after the last element of a reverse iteration
c.crbegin()	Returns a constant reverse iterator for the first element of a reverse iteration (since C++11)
c.crend()	Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11)

6.1.5.2. test

```
#include <iostream>
#include <map>
#include <functional>
using namespace std;

int main()
{
    map<int,string > mis;
    mis = {
        pair<int,string>(1,"a"),
        pair<int,string>(2,"abc")
    };

    map<int,string>::iterator itr;
```

```
    for(itr = mis.begin();itr != mis.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }
    return 0;
}
```

6.1.6. Inserting and Removing Elements

6.1.6.1. api

Operation	Effect
c.insert(val)	Inserts a copy of val and returns the position of the new element and, for maps, whether it succeeded
c.insert(pos,val)	Inserts a copy of val and returns the position of the new element (pos is used as a hint pointing to where the insert should start the search)
c.insert(beg,end)	Inserts a copy of all elements of the range [beg,end) (returns nothing)
c.insert(initlist)	Inserts a copy of all elements in the initializer list initlist (returns nothing; since C++11)
c.emplace(args...)	Inserts a copy of an element initialized with args and returns the position of the new element and, for maps, whether it succeeded (since C++11)
c.emplace_hint(pos,args...)	Inserts a copy of an element initialized with args and returns the position of the new element (pos is used as a hint pointing to where the insert should start the search)
c.erase(val)	Removes all elements equal to val and returns the number of removed elements
c.erase(pos)	Removes the element at iterator position pos and returns the following position (returned nothing before C++11)
c.erase(beg,end)	Removes all elements of the range [beg,end) and returns the following position (returned nothing before C++11)
c.clear()	Removes all elements (empties the container)

6.1.6.2. api 解析

移除某个 map 中某个条目用 erase（），该成员方法的定义如下:

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

mp.clear()就相当于 mp.erase(mp.begin(),mp.end());

6.1.6.3. test

```
#include <iostream>
#include <map>
#include <functional>
using namespace std;

int main()
{
    map<int,string > mis;
    mis.insert(pair<int,string>(1,"china"));
    mis.insert(map<int,string >::value_type(2,"canada"));
    mis.insert(make_pair(3,"america"));
    mis.insert(make_pair(3,"japan"));

    mis.emplace(4,"brazil");
    mis.emplace(5,"france");

    mis.insert({make_pair(6,"german"),make_pair(7,"germany")});

    map<int,string>::iterator itr;
    for(itr = mis.begin();itr != mis.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }

    mis.erase(7);
    for(itr = mis.begin();itr != mis.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }

    auto find = mis.find(6);
    if(find != mis.end())
        mis.erase(find);
    for(itr = mis.begin();itr != mis.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }

    mis.clear();


    return 0;
}
```

6.1.6.4. 小结

```
第一种：用 insert 函数插入 pair 数据：
```

```
    mapStudent.insert(pair<int, string>(1, "student_one"));
第二种：用 insert 函数插入 value_type 数据：
    mapStudent.insert(map<int, string>::value_type (1, "student_one"));
第三种：用数组方式插入数据。
    mapStudent[1] = "student_one";
```

当然了第一种和第二种在效果上是完成一样的，用 insert 函数插入数据，在数据的 插入上涉及到集合的唯一性这个概念，即当 map 中有这个关键字时，insert 操作是插入数据不了的，但是用数组方式就不同了，它可以覆盖以前该关键字对 应的值，用程序说明

6.1.7. Special Search Operations

6.1.7.1. api

Operation	Effect
c.count(val)	Returns the number of elements with key val
c.find(val)	Returns the position of the first element with key val (or end() if none found)
c.lower_bound(val)	Returns the first position where an element with key val would get inserted (the first element with a key >= val)
c.upper_bound(val)	Returns the last position where an element with key val would get inserted (the first element with a key > val)
c.equal_range(val)	Returns a range with all elements with a key equal to val (i.e., the first and last positions, where an element with key val would get inserted)

6.1.7.2. api 解析

lower_bound 函数用法，这个函数用来返回要查找关键字的下界(是一个迭代器)

upper_bound 函数用法，这个函数用来返回要查找关键字的上界(是一个迭代器)

例如：map 中已经插入了 1，2，3，4 的话，如果 lower_bound(2)的话，返回的 2，而 upper-bound（2）的话，返回的就是 3

equal_range 函数返回一个 pair，pair 里面第一个变量是 lower_bound 返回的迭代器，pair 里面第二个迭代器是 upper_bound 返回的迭代器，如果这两个迭代器相等的话，则说明 map 中不出现这个关键字，

6.1.7.3. test

```
#include <iostream>
#include <map>
#include <functional>
using namespace std;

int main()
{
   map<int,string > mis;
   mis = {
```

```
        pair<int,string>(1,"a"),
        pair<int,string>(2,"abc"),
        pair<int,string>(3,"abcd"),
        pair<int,string>(4,"abcde"),
        pair<int,string>(5,"abcdef"),
        pair<int,string>(6,"abcdefg"),
        pair<int,string>(7,"abcdefgh"),
        pair<int,string>(8,"abcdefghi"),
        pair<int,string>(9,"abcdefghij"),
        pair<int,string>(10,"abcdefghijk")

    };

    map<int,string>::iterator itr;
    for(itr = mis.begin();itr != mis.end(); ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }

    cout<<mis.count(1);cout<<endl;

    auto findItr = mis.find(40);
    if(findItr != mis.end())
        cout<<"find"<<endl;
    else
        cout<<"find none"<<endl;

    auto lowBound = mis.lower_bound(1);
    auto upperBound = mis.upper_bound(5);

    for(auto itr = lowBound; itr != upperBound; ++itr)
    {
        cout<<itr->first<<":"<<itr->second<<endl;
    }

    auto range  = mis.equal_range(3);
    pair<map<int,string>::iterator,map<int,string>::iterator>
            range2 = mis.equal_range(3);
     cout<<range2.first->first<<":"<<range2.second->first<<endl;

    return 0;
}
```

6.1.7.4. 小结

第一种：用 count 函数来判定关键字是否出现，其缺点是无法定位数据出现位置,由于 map 的特性，一对一的映射关系，就决定了 count 函数的返回值只有两个，要么是 0，要

么是 1，出现的情况，当然是返回 1 了。

第二种：用 find 函数来定位数据出现位置，它返回的一个迭代器，当数据出现时，它返回数据所在位置的迭代器，如果 map 中没有要查找的数据，它返回的迭代器等于 end 函数返回的迭代器。

查找 map 中是否包含某个关键字条目用 find()方法，传入的参数是要查找的 key，在这里需要提到的是 begin()和 end()两个成员，分别代表 map 对象中第一个条目和最后一个条目，这两个数据的类型是 iterator。

6.1.8. 补充

map 中的元素是自动按 Key 升序排序，所以不能对 map 用 sort 函数；

STL 中默认是采用小于号来排序的，基本数据类型在排序上是不存在任何问题的，因为它本身支持小于号运算，在一些特殊情况，比如关键字是一个结构体，涉及到排序就会出现问题，因为它没有小于号操作，insert 等函数在编译的时候过不去，就需要给出解决方案。

常用的解决方案：小于号重载，仿函数，lambda

6.2.multimap

允许 key 重复的 map，操作规则同上。

6.2.1. equal_range

```
#include <map>
using namespace  std;

int main ()
{
    multimap<char,int> mymm;

    mymm.insert(pair<char,int>('a',10));
    mymm.insert(pair<char,int>('b',20));
    mymm.insert(pair<char,int>('b',30));
    mymm.insert(pair<char,int>('b',40));
    mymm.insert(pair<char,int>('c',50));
    mymm.insert(pair<char,int>('c',60));
    mymm.insert(pair<char,int>('d',60));

    cout << "mymm contains:\n";
    for (char ch='a'; ch<='d'; ch++)
    {
       pair<multimap<char,int>::iterator, multimap<char,int>::iterator> ret;
       ret = mymm.equal_range(ch);
       cout << ch << " =>";
```

```
      for (multimap<char,int>::iterator it=ret.first; it!=ret.second;
++it)
         cout << ' ' << it->second;
      cout << '\n';
   }

   return 0;
}
```

6.3.set

仅有 key 且 key 不能重复的 map，操作规则同上。

6.4.multiset

6.4.1. range of multiset

```
#include <iostream>
#include <set>
using namespace std;

int main ()
{
   multiset<int> c;
   c.insert(1);
   c.insert(2);   c.insert(2);  c.insert(2);
   c.insert(4);   c.insert(4);
   c.insert(5);
   c.insert(6);
   cout << "lower_bound(3): " << *c.lower_bound(2) << endl;
   cout << "upper_bound(3): " << *c.upper_bound(2) << endl;
   cout << "equal_range(3): " << *c.equal_range(2).first << " "
        << *c.equal_range(3).second << endl;
   cout << endl;
   cout << "lower_bound(5): " << *c.lower_bound(2) << endl;
   cout << "upper_bound(5): " << *c.upper_bound(2) << endl;
   cout << "equal_range(5): " << *c.equal_range(2).first << " "
        << *c.equal_range(5).second << endl;

   return 0;
}
```

7. 无序 Unordered 关联容器

7.1.hash

7.1.1. hash 原理

Hash，一般翻译做“散列”，也有直接音译为“哈希”的，就是把任意长度的输入（又叫做预映射 pre-image）通过散列算法变换成固定长度的输出，该输出就是散列值。这种转换是一种压缩映射，也就是，散列值的空间通常远小于输入的空间，不同的输入可能会散列成相同的输出，所以不可能从散列值来确定唯一的输入值。简单的说就是一种将任意长度的消息压缩到某一固定长度的消息摘要的函数。

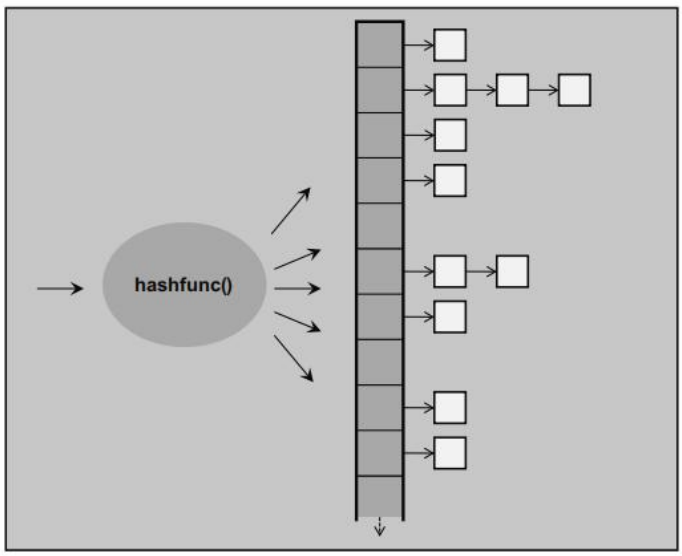
若结构中存在和关键字 K 相等的记录，则必定在 f(K)的存储位置上。由此，不需比较便可直接取得所查记录。称这个对应关系 f 为散列函数(Hash function)，按这个事先建立的表为散列表。

对不同的关键字可能得到同一散列地址，即 key1≠key2，而 f(key1)=f(key2)，这种现象称碰撞。具有相同函数值的关键字对该散列函数来说称做同义词。综上所述，根据散列函数 H(key)和处理冲突的方法将一组关键字映象到一个有限的连续的地址集（区间）上，并以关键字在地址集中的“象” 作为记录在表中的存储位置，这种表便称为散列表，这一映象过程称为散列造表或散列，所得的存储位置称散列地址。

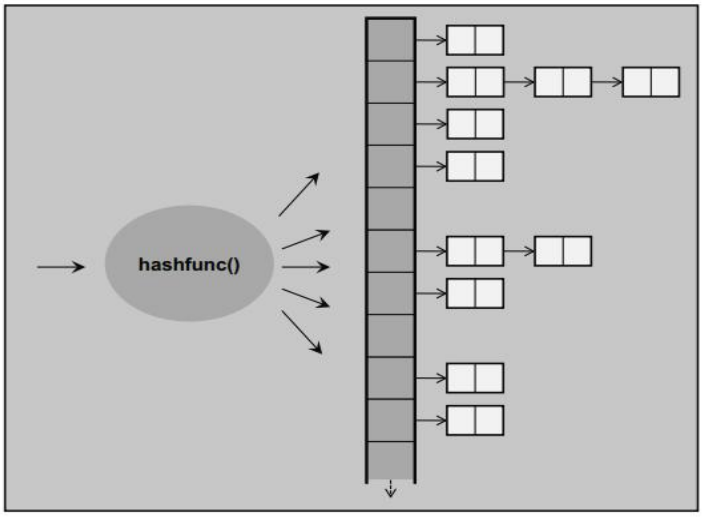
若对于关键字集合中的任一个关键字，经散列函数映象到地址集合中任何一个地址的概率是相等的，则称此类散列函数为均匀散列函数(Uniform Hash function)，这就是使关键字经过散列函数得到一个“随机的地址”，从而减少冲突。

7.1.2. hash-in-STL

7.1.2.1. hash_set



7.1.2.2. hash_map



7.1.3. 原理实战

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include <unordered_set>

using namespace std;

int main()
{
    unordered_set<string> coll;

    char buf[10];

    clock_t  startTime  = clock();
    for(int i=0; i<100000; i++)
    {
        snprintf(buf,10,"%d",rand());
        coll.insert(string(buf));
    }

    cout<<"milli-seconds: "<<(clock() - startTime)<<endl;
    cout<<"unordered_set.size(): "<<
         coll.size()<<endl;
    cout<<"unordered_set.bucket_count(): "<<
         coll.bucket_count()<<endl;
    cout<<"unordered_set.load_factor(): "<<
         coll.load_factor()<<endl;
    cout<<"unordered_set.max_load_factor(): "<<
```

```
        coll.max_load_factor()<<endl;
    cout<<"unordered_set.max_bucket_count(): "<<
        coll.max_bucket_count()<<endl;

    for(int i=0; i<30; i++)
    {
       cout<<"bucket # "<<i<<
            "has "<<coll.bucket_size(i)<<
            " elements"<<endl;
    }

    return 0;
}
```

7.2.unordered_map/multimap(C++ 11)

C++ 11 标准中加入了 unordered 系列的容器。unordered_map 记录元素的 hash 值，根据 hash 值判断元素是否相同。

7.2.1. unordered_map 与 map 的对比：

存储时是根据 key 的 hash 值判断元素是否相同，即 unordered_map 内部元素是无序的，而 map 中的元素是按照二叉搜索树存储（用红黑树实现），进行中序遍历会得到有序遍历。所以使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些。

总结：结构体用 map 重载<运算符，结构体用 unordered_map 重载==运算符。

7.2.2. unordered_map 模板

```
template < class Key,
         class T,
         class Hash = hash<Key>,
         class Pred = equal_to<Key>,
         class Alloc = allocator< pair<const Key,T> >
         > class unordered_map;
```

7.2.3. bucket

7.2.3.1. 原型

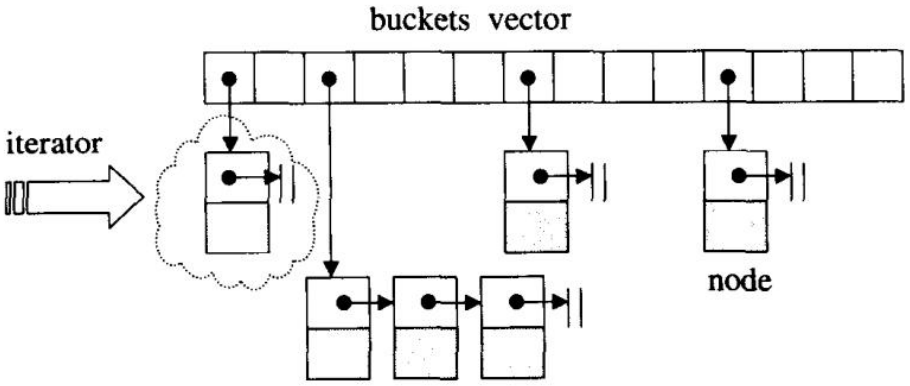
```
size_type bucket ( const key_type& k ) const;
```

7.2.3.2. 说明

定位元素所在的桶，返回 Key 值为输入参数 k 的元素的所在桶号。

桶是容器内部 Hash 表中的一个槽，槽中的元素根据 Key 值分配元素。桶号的编号从 0 到 (bucket_count - 1)。

桶中单个元素可以通过 unordered_map::begin 和 unordered_map::end 返回的范围迭代器进行访问。



7.2.3.3. 程序

```
for ( unsigned i = 0; i < mss.bucket_count(); ++i) {
    cout << "bucket #" << i << " contains:";
    for ( auto itr = mss.begin(i); itr!= mss.end(i); ++itr )
       cout << " " << itr->first << ":" << itr->second;
    cout << endl;
}
```

7.2.4. 测试

```
#include <iostream>
#include <unordered_map>
using namespace  std;
int main ()
{
   unordered_map<string,string> mss;
   mss = {
      {"Australia","Canberra"},
      {"U.S.","Washington"},
      {"France","Paris"}
   };

   cout << "mss contains:";
   for ( auto it = mss.begin(); it != mss.end(); ++it )
      cout << " " << it->first << ":" << it->second;
   cout << endl;

   cout << "mss's buckets contain:\n";
```

```
    for ( unsigned i = 0; i < mss.bucket_count(); ++i) {
        cout << "bucket #" << i << " contains:";
        for ( auto itr = mss.begin(i); itr!= mss.end(i); ++itr )
            cout << " " << itr->first << ":" << itr->second;
        cout << endl;
    }
    return 0;
}
```

7.3.unordered_set/multiset

8. 迭代器 Iterator

8.1.what

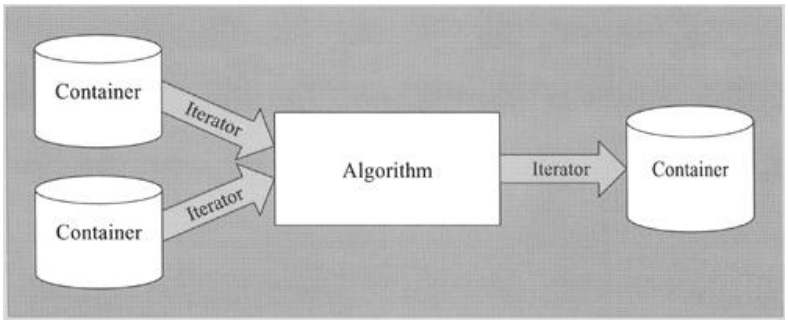
Iterator（迭代器）模式又称 Cursor（游标）模式，用于提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。或者这样说可能更容易理解：Iterator 模式是运用于聚合对象的一种模式，通过运用该模式，使得我们可以在不知道对象内部表示的情况下，按照一定顺序（由 iterator 提供的方法）访问聚合对象中的各个元素。

8.2.why

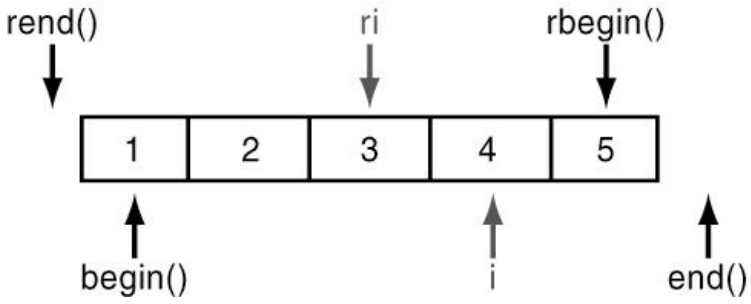
8.2.1. 功能

- 能够让迭代器与算法不干扰的相互发展，最后又能无间隙的粘合起来；
- 重载了 -> ,* ,++ ,--, ==, !=, == 运算符。用以操作复杂的数据结构；
- 容器提供迭代器，算法使用迭代器；

8.2.2. 图示



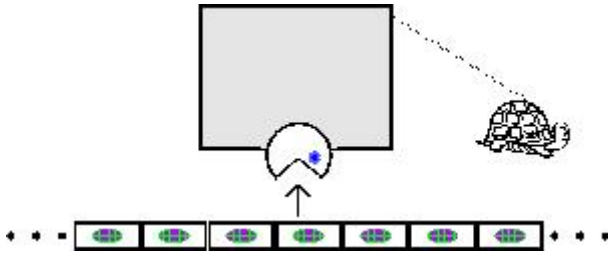
8.2.3. begin()/end()



8.3.Category

8.3.1. Input Iterators

8.3.1.1. 图示



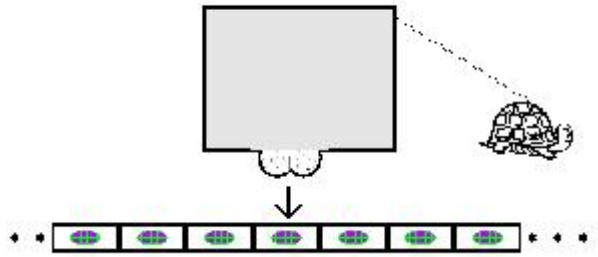
8.3.1.2. 计算

input 迭代器只能一次一个向前读取元素，按此顺序一个个传回元素值。input 迭代器的各项操作:

*iter	读取实际元素
iter->member	读取实际元素的成员
++iter	向前步进，传回新位置
iter++	向前步进，传回旧位置
iter1 == iter2	判断两个迭代器是否相等
TYPE(iter)	复制迭代器（copy 构造函数）

8.3.2. Output Iterators

8.3.2.1. 图示



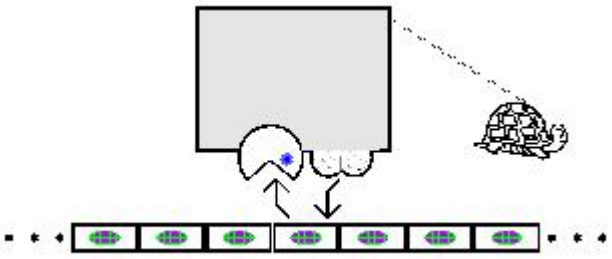
8.3.2.2. 计算

output 迭代器哦和 input 迭代器相反，其作用是将元素值一个个写入。也就是说，你只能一个元素一个元素地赋新值。output 相关操作:

*iter = value	将元素值写到迭代器位置
++iter	向前步进，传回新位置
iter++	向后步进，返回旧位置
TYPE(iter)	复制迭代器（copy 构造函数）

8.3.3. Forward Iterators

8.3.3.1. 图示



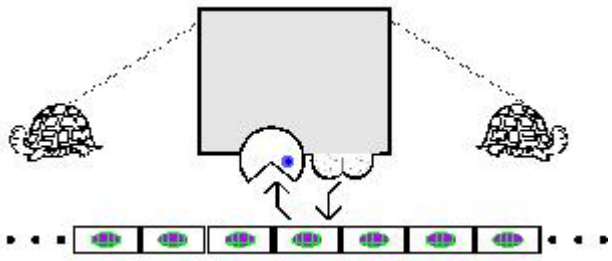
8.3.3.2. 计算

forword 迭代器是 input 和 output 迭代器的结合，兼具了两者的功能，更为普适的迭代器。forword 迭代器的各项操作:

*iter	存取实际元素
iter->member	读取实际元素的成员
++iter	向前步进，传回新位置
iter++	向前步进，传回旧位置
iter1 == iter2	判断两个迭代器是否相等
TYPE(iter)	复制迭代器（copy 构造函数）
TYPE()	产生迭代器（default 构造函数）
iter1 = iter2	赋值

8.3.4. Bidirectional Iterators

8.3.4.1. 图示



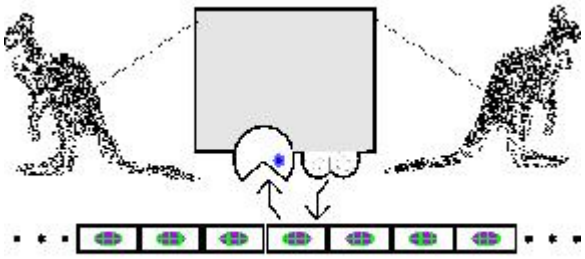
8.3.4.2. 计算

bidirectional 迭代器在 forword 迭代器的基础上增加了回头遍历的能力。换言之，它支持一步步后退的操作。**新增操作**:

--iter	退步（传回新位置）
iter--	退步（传回旧位置）

8.3.5. Random-Access Iterators

8.3.5.1. 图示



8.3.5.2. 计算

random access 迭代器，**新增**操作:

iter[n]	取索引位置为 n 的元素
iter += n	向前跳 n 个元素
iter -= n	向后跳 n 个元素
iter + n	传回 iter 之后的第 n 个元素
n + iter	传回 iter 之后的第 n 个元素
iter - n	传回 iter 之前的第 n 个元素
iter1 - iter2	传回 iter1 和 iter2 之间的距离
iter1 < iter2	判断 iter1 是否在 iter2 之前
iter1 > iter2	判断 iter1 是否在 iter2 之后
iter1 <= iter2	判断 iter1 是否不在 iter2 之后
iter1 >= iter2	判断 iter1 是否不在 iter2 之前

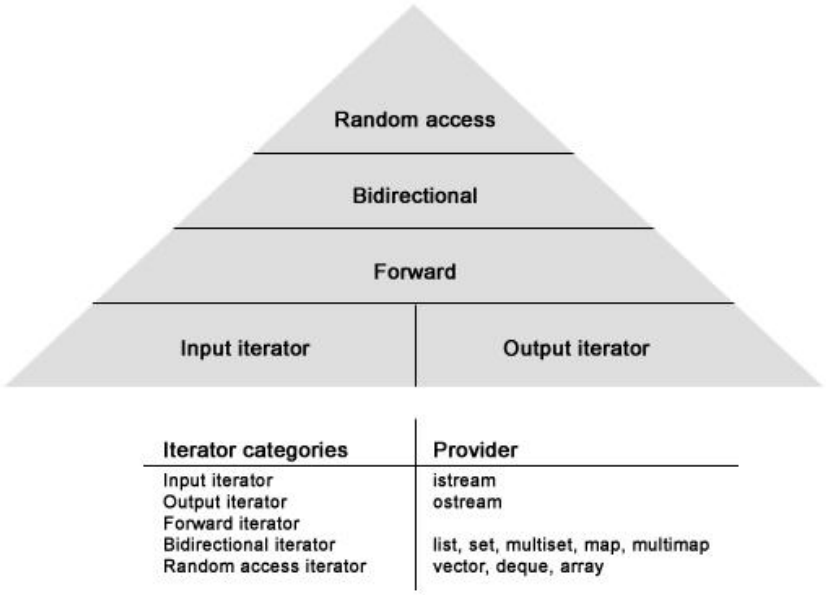
8.4.常见容器的迭代器

8.4.1. Category

容器	支持的迭代器类别
vector	随机访问
deque	随机访问
list	双向

set	双向
multiset	双向
map	双向
multimap	双向
stack	不支持
queue	不支持
priority_queue	不支持

8.4.2. Diagram



8.5.迭代器附加 Auxiliary Iterator Functions

8.5.1. introduction

STL 提供了额外的 5 个函数，advance(),next(), prev(), distance(), and iter_swap(),前 4 个函数，是用于非 random-access Iterator，可以实现多步跨越。最后一个函数用于交换两个 iterator。

#include <iterator>
void **advance** (InputIterator& pos, Dist n)
ForwardIterator **next** (ForwardIterator pos) ForwardIterator **next** (ForwardIterator pos, Dist n)
BidirectionalIterator **prev** (BidirectionalIterator pos) BidirectionalIterator **prev** (BidirectionalIterator pos, Dist n)
Dist **distance** (InputIterator pos1, InputIterator pos2)
void **iter_swap** (ForwardIterator1 pos1, ForwardIterator2 pos2)

8.5.2. more details

请自行测之。

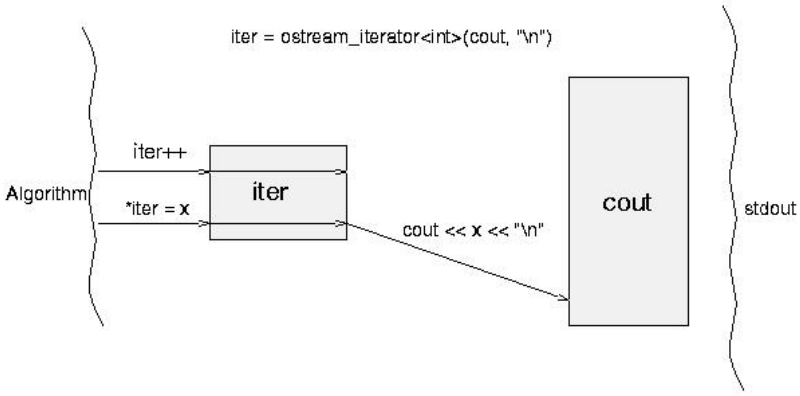
8.6.Iterator adapter

8.6.1. Reverse Iterators 反向迭代器

8.6.2. Insert Iterators 插入迭代器

8.6.3. Stream Iterators 流迭代器

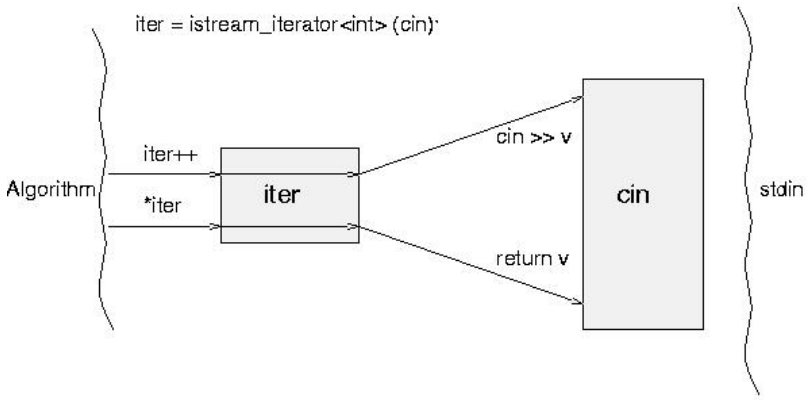
8.6.3.1. ostream_iterator




```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;
int main()
{

    ostream_iterator<int> intWriter(cout,"\n");

    *intWriter = 42;
    intWriter++;
    *intWriter = 77;
    intWriter++;
    *intWriter = -5;


    vector<int> coll = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // write all elements without any delimiter
    copy (coll.cbegin(), coll.cend(),
```

```
        ostream_iterator<int>(cout));
    cout << endl;
    // write all elements with " < " as delimiter
    copy (coll.cbegin(), coll.cend(),
        ostream_iterator<int>(cout," < "));
    cout << endl;
}
```

8.6.3.2. istream_iterator




```
#include <iostream>
#include <iterator>
using namespace std;
int main()
{

    istream_iterator<int> intReader(cin);

    istream_iterator<int> iiEOF;

    while (intReader != iiEOF) {
        cout << "once: " << *intReader << endl;
        ++intReader;
    }
}
```

8.7.invalid Iterator 失效迭代器

8.7.1. vector<>::iterator

8.7.2. list<>::iterator

8.7.3. map<>::iterator

9. 仿函数 Functor

仿函数(functor)又称之为函数对象(function object),其实就是重载了()操作符的 class,对象名可以像函数名一样使用。

9.1.自实现 self-defined functor

9.1.1. grammar

```
class FunctionObjectType
{
public:
    void operator() ()
    {
       ........
    }
};
```

9.1.2. e.g.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

class avg
{
public:
    avg(int n=0,int s=0)
       :num(n),sum(s){}
    void operator()(int e)
    {
       ++num;
       sum += e;
    }
    double data()
    {
       return (double)sum/num;
    }

protected:
    int num;
    int sum;
};
```

```
int main()
{

    vector<int> vi =  {1,3,5,6,7,2,4,6,8,10};
    avg mean = for_each(vi.begin(),vi.end(),avg());
    cout<<mean.data();
    return 0;
}
```

9.2.STL 内置 functor

要使用 STL 内建的仿函数，必须包含<functional>头文件。而头文件中包含的仿函数分类包括。

9.2.1. 分类 Category

关于其分类，通常两种方式，一种是按参数的个数进行分类，分为 unary_function 和 binary_function。另外一种，就是按其功能进行分类。比如下介绍，就是按其功能进行分类的。

unary 和 binary 是可以通过函数的语义获知的。比如遍历中的 functor 就是 unary 的，而排序中的 fucntor 就是 binary。

9.2.2. 数学相关 Arithmetic

9.2.2.1. Collection

函数对象类模板	成员函数 T operator (const T & x, const T & y) 的功能
plus <T>	return x + y;
minus < >	return x - y;
multiplies <T>	return x * y;
divides <T>	return x / y;
modulus <T>	return x % y;
	成员函数 bool operator(const T & x, const T & y) 的功能
equal_to <T>	return x == y;
not_equal_to <T>	return x! = y;

greater <T>	return x > y;
less <T>	return x < y;
greater_equal <T>	return x > = y;
less_equal <T>	return x <= y;
logical_and <T>	return x && y;
logical_or <T>	return x || y;
negate <T>	return - x;
logical_not <T>	return ! x;

negate<*type*>()	*- param*
plus<*type*>()	*param1 + param2*
minus<*type*>()	*param1 - param2*
multiplies<*type*>()	*param1 * param2*
divides<*type*>()	*param1 / param2*
modulus<*type*>()	*param1 % param2*

9.2.2.2. e.g.

```
#include <functional>
#include <iostream>
using namespace std;

int main()
{
    auto plus10 = bind(
              plus<int>(),
              placeholders::_1,
              10);
    cout << "+10: " << plus10(7) << endl;

    auto plus10times2 = bind(
              multiplies<int>(),
              bind(
                  plus<int>(),
                  placeholders::_1,
                  10),
              2);
    cout << "+10 *2: " << plus10times2(7) << endl;

    auto pow3 = bind(
              multiplies<int>(),
```

```
            bind(multiplies<int>(),
                    placeholders::_1,
                    placeholders::_1),
            placeholders::_1);

    cout << "x*x*x: " << pow3(7) << endl;

    auto inversDivide = bind(
            divides<double>(),
            placeholders::_2,
            placeholders::_1);
    cout << "invdiv: " << inversDivide(49,7) << endl;
    return 0;
}
```

9.2.3. 关系相关 Relational

9.2.3.1. Collection

equal_to<*type*>()	*param1* == *param2*
not_equal_to<*type*>()	*param1* != *param2*
less<*type*>()	*param1* < *param2*
greater<*type*>()	*param1* > *param2*
less_equal<*type*>()	*param1* <= *param2*
greater_equal<*type*>()	*param1* >= *param2*

9.2.3.2. e.g.

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
#include <iterator>

using namespace std;


int main()
{
    vector<int> vi = {1,3,5,7,9,2,4,6,8,10};

    sort(vi.begin(),vi.end(),less<int>());

    for(auto &i: vi)
        cout<<i<<" ";

    cout<<endl;
```

```
    int count =
    count_if(vi.begin(),vi.end(),bind(greater<int>(),std::placeholders::_1,5));
    cout<<count<<endl;

    return 0;
}
```

9.2.4. 逻辑相关 Logical

9.2.4.1. Collection

logical_not<*type*>()	! *param*
logical_and<*type*>()	*param1* && *param2*
logical_or<*type*>()	*param1* || *param2*

9.2.4.2. e.g.

9.3.适配与绑定 functor adapters && binders

9.3.1. Category

Operation	Effect
bind(op,args...)	Binds args to op
~~mem_fn(op)~~	~~Calls op() as a member function for an object or pointer to object~~
not1(op)	Unary negation: !op(param)
not2(op)	Binary negation: !op(param1,param2)

9.3.2. bind1st/bind2nd

std::for_each() 要求它的第三个参数是一个仅接受正好一个参数的函数或函数对象。如果 std::for_each() 被执行，指定容器中的所有元素，这些元素的类型为 int - 将按顺序被传入至 print() 函数。

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

void print(int i)
{
    cout << i << endl;
}
int main()
{
    vector<int>  vi = {1,3,5,7,9,2,4,6,8,10};
    for_each(vi.begin(),vi.end(),print);
```

```
    return 0;
}
```

但是，如果要使用一个具有不同签名的函数的话，事情就复杂了。 例如，如果要传入的是以下函数 add()，它要将一个常数值加至容器中的每个元素上，并显示结果。

```
void add(int i, int j)
{
    std::cout << i + j << std::endl;
}
int main()
{
    vector<int>  vi = {1,3,5,7,9,2,4,6,8,10};
    for_each(vi.begin(),vi.end(),bind1st(add,10));
    return 0;
}
```

add() 函数已被转换为一个派生自 std::binary_function 的函数对象。如果我们不修改 add 为函数对象的话而直接将 add()自由函数传给 bind1st，编译器将会报错。

```

class add:public binary_function<int,int,void >
{
public:
    void operator()(int i, int j) const
    {
        std::cout << i + j << std::endl;
    }
};
int main()
{
    vector<int>  vi = {1,3,5,7,9,2,4,6,8,10};
    for_each(vi.begin(),vi.end(),bind1st(add(),10));
    return 0;
}
```

9.3.3. bind

bind 简化了不同函数之间的绑定,从而取代了 1st 和 2nd。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>
using namespace  std;
using namespace  std::placeholders;
```

```
void add(int i, int j)
{
    cout << i + j << endl;
}

int main()
{
    std::vector<int> v;
    v.push_back(1);
    v.push_back(3);
    v.push_back(2);

    for_each(v.begin(), v.end(), bind(add,10,_1));
    return 0;
}
```

像 add() 这样的函数不再需要为了要用于 std::for_each() 而转换为函数对象。 使用 boost::bind()，这个函数可以忽略其第一个参数而使用。

因为 add() 函数要求两个参数，两个参数都必须传递给 bind()。 第一个参数是常数值 10，而第二个参数则是一个怪异的_1 占位。

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;
using namespace std::placeholders;

void add(int i, int j)
{
    cout << i + j << endl;
}
bool compare(int i, int j)
{
    return i > j;
}
void printVector(vector<int> &v)
{
    for(vector<int>::iterator i = v.begin(); i != v.end(); i++)
        cout << *i << " " ;
    cout << endl;
}
int main()
{
    vector<int> v;
    v.push_back(1);
    v.push_back(3);
```

```
    v.push_back(2);

    //for_each(v.begin(), v.end(), bind1st(add,10));

    cout << "? ? ? ? ? ? ? ? ? " << endl;
    for_each(v.begin(), v.end(), bind(add,19,_1));

    cout << "? ? ? ? ? ? ? ? ? " << endl;
    sort(v.begin(), v.end(), bind(compare,_1,_2));
    printVector(v);

    cout << "? ? ? ? ? ? ? ? ? " << endl;
    sort(v.begin(), v.end(), compare);
    printVector(v);

    cout << "? ? ? ? ? ? ? ? ? " << endl;
    sort(v.begin(), v.end(), bind(compare,_2,_1));
    printVector(v);
    return 0;
}
```

综合练习:

```
#include <algorithm>
#include <vector>
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;


int main()
{
    vector<int> vi(10);

    iota(vi.begin(),vi.end(),6);

    //大于 10 的数据
    int count = count_if(vi.begin(),
                         vi.end(),
                         bind1st(less<int>(),10));

    cout<<"count:"<<count<<endl;

    //小于 10 的数据
    count  = count_if(vi.begin(),
                      vi.end(),
                      bind2nd(less<int>(),10));
```

```
    //大于 10 的数据
    count = count_if(vi.begin(),vi.end(),
                  bind(less<int>(),10,_1));

    //小于 10 的数据
    count = count_if(vi.begin(),vi.end(),
                  bind(less<int>(),_1,10));

    return 0;
}
```

9.3.4. not1/not2 与 bind1st/bind2nd

9.3.5. bind 绑定分类

用于绑定可调用对象和参数使用。

9.3.5.1. 绑定普通可调用对象

9.3.5.2. 绑定类成员引入

```
#include <iostream>
#include <functional>
using namespace std;
using namespace std::placeholders;

class A
{
public:
    void print(int x, int y)
    {
        cout<<"x:"<<x<<endl;
        cout<<"y:"<<y<<endl;
    }
};

int main()
{
    A a;
    auto  f = bind(&A::print,a,1,2);
    f();
    auto f1 = bind(&A::print,::_1,_2,_3);
    f1(a,3,5);
    auto  f2 = bind(&A::print,_1,100,200);
```

```
    f2(a);
    auto  f3 = bind(&A::print,a,_1,_2);
    f3(333,444);
    return 0;
}
```

9.3.5.3. 绑定类成员函数

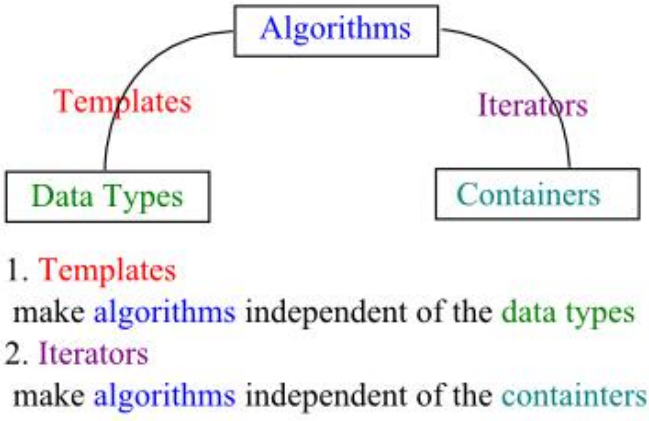
```
#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>
#include <string>
using namespace std;
using namespace std::placeholders;
class Person
{
private:
    string name;
public:
    Person (const string& n) : name(n)
    {
    }
    void print () const
    {
        cout << name << endl;
    }

    void print2 (const string& prefix) const
    {
        cout << prefix << name << endl;
    }

};
int main()
{

    Person p("abc");
    auto mem = bind(&Person::print,p);
    mem();

#if 0

    vector<Person> vp = {
         Person("Tick"),
         Person("Trick"),
         Person("Track")
```

```
    };

    for_each (vp.begin(), vp.end(),
            bind(&Person::print,_1));
    cout << endl;

    for_each (vp.begin(), vp.end(),
            bind(&Person::print2,_1,"Person: "));
    cout << endl;

    bind(&Person::print2,_1,"This is: ")(Person("nico"));

#endif

    return 0;
}
```

10.算法 Algorithm

10.1.综述

STL 算法部分主要由头文件<algorithm>,<numeric>,<functional>组成。要使用 STL 中的算法函数必须包含头文件<algorithm>，对于数值算法须包含<numeric>，<functional>中则定义了一些模板类，用来声明函数对象。



如果有**同名**的成员算法函数，优先选用成员算法函数，为什么呢，因为这样，可以获得更高的效率。

10.2.Category

详见附录 A

10.3.排序

10.3.1. 声明及语义

函数名	函数功能
sort	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
	template<class RanIt> void sort(RanIt first, RanIt last);
	template<class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);
stable_sort	与 sort 类似，不过保留相等元素之间的顺序关系
	template<class BidIt> void stable_sort(BidIt first, BidIt last);
	template<class BidIt, class Pred> void stable_sort(BidIt first, BidIt last, Pred pr);
partition	对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前
	template<class BidIt, class Pred> BidIt partition(BidIt first, BidIt last, Pred pr);
stable_partition	与 partition 类似，不过不保证保留容器中的相对顺序
	template<class FwdIt, class Pred> FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);
random_shuffle	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
	template<class RanIt> void random_shuffle(RanIt first, RanIt last);
	template<class RanIt, class Fun> void random_shuffle(RanIt first, RanIt last, Fun& f);

10.3.2. 实战

10.3.2.1. sort

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void print(int &v)
{
    cout<<v<<" ";
}
int main()
{
    vector<int> vi;
    for(int i=0; i<5; i++) vi.push_back(i);

//    sort(vi.begin(),vi.end(),greater<int>());
```

```
//    sort(vi.begin(),vi.end(),less_equal<int>());
    stable_sort(vi.begin(),vi.end(),greater_equal<int>());

    for_each(vi.begin(),vi.end(),print);
    return 0;
}
```

10.3.2.2. partition

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

bool isodd(int i )
{
    return (i%2) == 1;
}


int main()
{
    int data[10]={3,6,9,2,5,8,1,4,7,0};
    vector<int> vi(data,data+10);


//    partition(vi.begin(),vi.end(),isodd);
    stable_partition(vi.begin(),vi.end(),isodd);

    for(auto i:vi)
    {
        cout<<i<<endl;
    }

    return 0;
}
```

10.3.2.3. random

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>
```

```
int myrandom (int i)
{
   return std::rand()%i;
}

int main ()
{
  std::srand ( unsigned ( std::time(0) ) );
  std::vector<int> myvector;


  for (int i=1; i<10; ++i) myvector.push_back(i);
    // 1 2 3 4 5 6 7 8 9


//  std::random_shuffle ( myvector.begin(), myvector.end() );


  std::random_shuffle ( myvector.begin(),
                                      myvector.end(), myrandom);


  std::cout << "myvector contains:";
  for (auto it: myvector)
    std::cout << ' ' << it;

  std::cout << '\n';

  return 0;
}
```

10.4.查找

10.4.1. 声明及语义

函数名	函数功能
find	利用底层元素的等于操作符,对指定范围内的元素与输入值进行比较.当匹配时,结束搜索,返回该元素的一个 InputIterator
	template<class InIt, class T> InIt find(InIt first, InIt last, const T& val);
find_if	使用输入的函数代替等于操作符，执行 find
	template<class InIt, class Pred> InIt find_if(InIt first, InIt last, Pred pr);
count	利用等于操作符,把标志范围内的元素与输入值比较,返回相等元素个数

	template<class InIt, class Dist> size_t count(InIt first, InIt last,const T& val, Dist& n);
count_if	利用输入的操作符，对标志范围内的元素进行操作，返回结果为 true 的个数
	template<class InIt, class Pred, class Dist> size_t count_if(InIt first, InIt last, Pred pr);
search **(范围查找)**	给出两个范围，返回一个 ForwardIterator,查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向 last1,重载版本使用自定义的比较操作
	template<class FwdIt1, class FwdIt2> FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2);
	template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);
search_n **(连续查找)**	在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作
	template<class FwdIt, class Dist, class T> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val);
	template<class FwdIt, class Dist, class T, class Pred> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val, Pred pr);

10.4.2. 实战

10.4.2.1. find

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
bool func(int & v)
{
    return v >= 3?true:false;
}
int main()
{
    int data[5] = {1,2,3,4,5};
    int *p = find(data,data+5,4);
    if(p != data+5)
        printf("%d\n",*p);
    else
        printf("find none\n");

    vector<int> vi(data,data+5);

    vector<int>::iterator itr;
    itr = find(vi.begin(),vi.end(),10);
    if(itr != vi.end())
```

```
        cout<<*itr<<endl;
    else
        cout<<"find none"<<endl;
    itr=find_if(vi.begin(),vi.end(),func);
    if(itr != vi.end())
        cout<<*itr<<endl;
    else
        cout<<"find none"<<endl;
    return 0;
}
```

10.4.2.2. count

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

bool func(int &v)
{
    return v > 10 ? true:false;
}

int main ()
{

    int myints[] = {10,20,30,30,20,10,10,20};
    int mycount = count (myints, myints+8, 10);
    cout << "10 appears " << mycount << " times.\n";


    vector<int> myvector (myints, myints+8);
    mycount = count (myvector.begin(), myvector.end(), 20);
    cout << "20 appears " << mycount  << " times.\n";


    mycount = count_if (myvector.begin(), myvector.end(), func);
    cout << ">10 appears " << mycount  << " times.\n";

    return 0;
}
```

10.4.2.3. search

```
#include <string>
#include <algorithm>
#include <iostream>
```

```
#include <vector>
using namespace std;

template<typename Container>
bool inQuote(const Container& cont, const string& s)
{
    return
       search(cont.begin(), cont.end(), s.begin(), s.end())
       != cont.end();
}

int main()
{
    string str =
    "why waste time learning, when ignorance is instantaneous?";

   cout << boolalpha << inQuote(str, "learning") << '\n'
            << inQuote(str, "lemming")  << '\n';

   vector<char> vec(str.begin(), str.end());
   cout << boolalpha << inQuote(vec, "learning") << '\n'
            << inQuote(vec, "lemming")  << '\n';
   return 0;
}
```

```
#include <iostream>
#include <algorithm>
#include <iterator>
using namespace std;

template <class Cont, class Size, class T>
bool consecutive_values(const Cont& c, Size count, const T& v)
{
   return search_n
          (begin(c),end(c),count,v)
          != end(c);
}

int main()
{
   const char sequence[] = "10010101000101010010101";

   cout << boolalpha;
   cout << "Has 4 consecutive zeros: "
            << consecutive_values(sequence,4,'0')<<endl;
   cout << "Has 3 consecutive zeros: "
            << consecutive_values(sequence,3,'0')<<endl;
```

```
    return 0;
}
```

10.5.删除与替换

10.5.1. 声明及语义

函数名	函数功能
copy	复制序列
	template<class InIt, class OutIt> OutIt copy(InIt first, InIt last, OutIt x);
remove	删除指定范围内所有等于指定元素的元素。注意，该函数**不是真正**删除函数。内置函数不适合使用 remove 和 remove_if 函数
	template<class FwdIt, class T> FwdIt remove(FwdIt first, FwdIt last, const T& val);
remove_if	删除指定范围内输入操作结果为 true 的所有元素
	template<class FwdIt, class Pred> FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);
remove_copy	将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置
	template<class InIt, class OutIt, class T> OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
remove_copy_if	将所有不匹配元素拷贝到一个指定容器
	template<class InIt, class OutIt, class Pred> OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
unique	清除序列中重复元素，和 remove 类似，它也不能真正删除元素。重载版本使用自定义比较操作
	template<class FwdIt> FwdIt unique(FwdIt first, FwdIt last);
	template<class FwdIt, class Pred> FwdIt unique(FwdIt first, FwdIt last, Pred pr);
unique_copy	与 unique 类似，不过把结果输出到另一个容器
	template<class InIt, class OutIt> OutIt unique_copy(InIt first, InIt last, OutIt x);
	template<class InIt, class OutIt, class Pred> OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);

10.5.2. 实战

10.5.2.1. copy

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <iterator>
```

```
#include <numeric>


using namespace std;

int main()
{

    vector<int> from = {1,2,3,4,5,6,7,8};

    vector<int> to(10);
    copy(from.begin(), from.end(),
            to.begin());
    to.resize((from.size()));

    cout << "to_vector contains: ";

    copy(to.begin(), to.end(),
            ostream_iterator<int>(cout, " "));
    cout << '\n';
}
```

10.5.2.2. remove

```
#include <algorithm>
#include <string>
#include <iostream>
#include <cctype>
using namespace std;

int main ()
{
  int myints[] = {10,20,30,30,20,10,10,20};
  // 10 20 30 30 20 10 10 20

  // bounds of range:
  int* pbegin = myints;

  int* pend = myints+sizeof(myints)/sizeof(int);


  pend = remove (pbegin, pend, 20);
  // 10 30 30 10 10 ? ? ?

  cout << "range contains:";
  for (int* p=pbegin; p!=pend; ++p)
    cout << ' ' << *p;
  cout << '\n';
```

```
  return 0;
}

int main2()
{
    string str1 = "Text with some   spaces";
    str1.erase(remove(str1.begin(), str1.end(), ' '),
               str1.end());
    cout << str1 << '\n';

    string str2 = "Text\n with\tsome \t  whitespaces\n\n";
    str2.erase(remove_if(str2.begin(),
                         str2.end(),
                         [](char x){return isspace(x);}),
               str2.end());
    cout << str2 << '\n';
}
```

10.5.2.3. unique

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>
#include <cctype>
using namespace std;

int main()
{

    vector<int> v = {1,2,3,1,2,3,3,4,5,4,5,6,7};
    sort(v.begin(), v.end());
    // 1 1 2 2 3 3 3 4 4 5 5 6 7
    auto last = unique(v.begin(), v.end());
    // v now holds {1 2 3 4 5 6 7 x x x x x x},
    // where 'x' is indeterminate
    v.erase(last, v.end());
    for (int i : v)
      cout << i << " ";
    cout << "\n";

    // remove consecutive spaces
    string s = "wanna go    to     space?";
    auto end = unique(s.begin(), s.end(),
                      [](char l, char r){
        return isspace(l) && isspace(r) && l == r;
    });
    // s now holds "wanna go to space?xxxxxxxx",
```

```
    // where 'x' is indeterminate
    cout << string(s.begin(), end) << '\n';

    return 0;
}
```

10.6.排列组合

10.6.1. 声明及语义

函数名	函数功能
next_permutation	取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作
	template<class BidIt> bool next_permutation(BidIt first, BidIt last);
	template<class BidIt, class Pred> bool next_permutation(BidIt first, BidIt last, Pred pr);
prev_permutation	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 false。重载版本使用自定义的比较操作
	template<class BidIt> bool prev_permutation(BidIt first, BidIt last);
	template<class BidIt, class Pred> bool prev_permutation(BidIt first, BidIt last, Pred pr);

10.6.2. 实战

10.6.2.1. next

```
#include <iostream>
#include <algorithm>

int main ()
{
    int myints[] = {1,2,3};

    std::sort (myints,myints+3);

    std::cout << "The 3! possible permutations with 3 elements:\n";
     do
     {
        std::cout << myints[0] << ' '
                  << myints[1] << ' '
                  << myints[2] << '\n';
    } while(std::next_permutation(myints,myints+3));

    std::cout << "After loop: "
```

```
            << myints[0] << ' '
            << myints[1] << ' '
            << myints[2] << '\n';

   return 0;
}
```

10.6.2.2. pre

```
#include <iostream>
#include <algorithm>

int main ()
{
   int myints[] = {1,2,3};

   std::sort (myints,myints+3);

   std::cout << "The 3! possible permutations with 3 elements:\n";
   do
   {
      std::cout << myints[0] << ' '
                << myints[1] << ' '
                << myints[2] << '\n';
   } while(std::pre_permutation(myints,myints+3));

   std::cout << "After loop: "
             << myints[0] << ' '
             << myints[1] << ' '
             << myints[2] << '\n';

   return 0;
}
```

10.7.集合

10.7.1. 声明及语义

函数名	函数功能
set_union	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_union(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2,OutIt x, Pred pr);

set_difference	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
set_ intersection	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2,InIt2 last2, OutIt x, Pred pr);

10.7.2. 实战

10.7.2.1. union

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main()
{
    int  first[5] = {1,2,5,7,9};
    int  second[5] = {1,2,6,8,10};

    vector<int> vi(20);
    vector<int>::iterator itr;

    sort(first,first+5);
    sort(second,second+5);

    itr = set_union(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());
    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

10.7.2.2. difference

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

using namespace std;

int main()
{
    int  first[5] = {1,2,5,7,9};
    int  second[5] = {1,2,6,7,10};

    vector<int> vi(20);
    vector<int>::iterator itr;

    sort(first,first+5);
    sort(second,second+5);

    itr = set_difference(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());

    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

10.7.2.3. intersection

```
#include <iterator>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main()
{
    int  first[5] = {1,2,5,7,9};
    int  second[5] = {1,2,6,8,10};

    vector<int> vi(20);
    vector<int>::iterator itr;

    sort(first,first+5);
```

```
    sort(second,second+5);

    itr = set_intersection(first,first+5,second,second+5,vi.begin());

    vi.resize(itr -vi.begin());


    copy(vi.begin(),vi.end(),ostream_iterator<int>(cout," "));

    return 0;
}
```

11.STL 线程安全性

STL 是**非线程安全**的，sgi 官言给出的解释和解决方案。https://www.sgi.com/tech/stl/thread_safety.html

12.项目实战

12.1.效率测试

12.1.1. 题目

设计一个测试程序，分别向 vector/list/map/multimap/set/multiset 中插入 1,000,000 条数据，测试插入时间，随机查找一个成员的，查找时间。

12.1.2. 时间计算

C++中的计时函数是 clock()，而与其相关的数据类型是 clock_t（头文件是 time.h）。函数定义原型为：clock_t clock(void);这个函数返回从"开启这个程序进程"到"程序中调用 clock()函数"时之间的 CPU 时钟计时单元（clock tick）数。

clock_t 是一个长整形数,另外在 time.h 文件中，还定义了一个常量 CLOCKS_PER_SEC，它用来表示一秒钟会有多少个时钟计时单元，因此，我们就可以使用公式 clock()/CLOCKS_PER_SEC 来计算一个进程自身的运行时间。

```
#include<iostream>
#include<ctime>
using namespace  std;
int main()
{
   clock_t start,finish;
   double totaltime;
   start=clock();

   int i = 0x7fffffff;
   while(i--);               //把你的程序代码插入到这里面

   finish=clock();
   totaltime=(double)(finish-start)/CLOCKS_PER_SEC;
   cout<<"\ntime interval :"<<totaltime<<" second !"<<endl;
}
```

12.2.海量搜索应用

12.2.1. 数据入库 txt -> sqlite3

现有某网站，数据库"脱裤子"数据，千万级别，格式如下。

现在要求对数据进行编辑查找操作，首先要对数据进行录入数据库，请依据给出的数据格式，设计程序将数据录入数据库。

12.2.2. sqlite3 -> STL

数据录入数据库以后，此时要检索某一类数据，并进行修改，设计程序，将数据库数据载入内存，完成相关的检索修改后，将数据回写到数据库。

```
#include <iostream>
#include <string.h>
#include <sqlite3.h>
#include <vector>
#include <map>
using namespace std;


void trimStrRightSpace(char *src)
{
   while(*src) ++src;
   while(*--src ==' ')
      *src = '\0';
}

void trimStrRightRetrun(char *src)
{
   while(*src) ++src;
   while(*--src =='\n')
      *src = '\0';
}

void trimStrLeftSpace(char *src)
{
   char *s = src;
   if(*src != ' ')
      return ;
   while(*src == ' ') ++src;
   while(*s++ = *src++);

}

void trimStrAllSpace(char *src)
{
   char * t = src;
   while(*t)
   {
```

```
        if(*t != ' ')
            *src++ = *t;

        ++t;
    }
    *src = '\0';
}


struct user
{
    char * email;
    char * name;
    char * addr;
    char * level;
    char * score;
    char * phone;
    char *mobile;
    char * money;
};

struct Person
{
    string email;
    string name;
    string addr;
    int    level;
    double score;
    string phone;
    string mobile;
    double money;
};


vector<Person> vp;

map<string,Person *> msvpi;


int callback(void *NotUsed, int argc, char **argv, char **ColName){

#if 0
    int i;
    for(i=0; i<argc; i++){
        printf("%s = %s\n", ColName[i], argv[i] ? argv[i] : "NULL");
    }
    printf("\n");
```

```
   static int flag = 1;
   if(flag)
   {
      for(int i=0; i<argc; i++)
      {
          printf("%-10s\t",ColName[i]);
      }
      flag = 0;
   }
   putchar(10);
#endif
   struct Person person;
   person.email = argv[1];
   person.name  = argv[2];
   person.addr  = argv[3];
   person.level = atoi(argv[4]);
   person.score = atof(argv[5]);
   person.phone = argv[6];
   person.mobile = argv[7];
   person.money = atof(argv[8]);
   vp.push_back(person);
   return 0;

}

int main()
{
    sqlite3 *db;
    char *zErrMsg = 0;
    int rc;

    rc = sqlite3_open("ddwdb", &db);  //如果数据库不存在，则创建，存在则打开。
    if( rc ){
       fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
       sqlite3_close(db);
       return(1);
    }

    char *sql = "select * from users where name = \"王桂林\"";
    rc = sqlite3_exec(db, sql, callback, 0, &zErrMsg);
    if( rc!=SQLITE_OK ){
       fprintf(stderr, "SQL error: %s\n", zErrMsg);
       sqlite3_free(zErrMsg);
    }

    //13522783198

    for(auto &p: vp)
```

```
    {
//       cout<<"email:"<<p.email<<"name"<<p.name<<"phone"<<p.phone<<endl;
        msvpi.insert(pair<string,Person *>(p.phone,&p));
    }

    auto itr = msvpi.find("13522783198");
    if(itr != msvpi.end())
        cout<<itr->second->addr<<endl;




#if 0

    char *finalsql = new char[1024*1024*30];
    char createSql[1024];

    strcpy(createSql,
"create table
users(id integer primary key autoincrement, email text,name text,addr
text,level integer, score real, phone text,mobile text,money real)");
    rc = sqlite3_exec(db, createSql, nullptr, 0, &zErrMsg);
    if( rc!=SQLITE_OK ){
        fprintf(stderr, "SQL error: %s\n", zErrMsg);
        sqlite3_free(zErrMsg);
    }


    FILE*  pf = fopen("dangdangwang.txt","r+");

    struct user  u;
    char buf[1024];
    char values[1024];
    int count = 0;
    while(fgets(buf,1024,pf))
    {

        trimStrAllSpace(buf);
        trimStrRightRetrun(buf);

        u.email  = strtok(buf,",");
        u.name   = strtok(nullptr,",");
        u.addr   = strtok(nullptr,",");
        u.level  = strtok(nullptr,",");
        u.score  = strtok(nullptr,",");
        u.phone  = strtok(nullptr,",");
        u.mobile = strtok(nullptr,",");
        u.money  = strtok(nullptr,",");
```

```
        if(count == 0)
        {
           memset(finalsql,0,1024*1024*3);
           strcpy(finalsql,"insert into users values");
        }

sprintf(values,"(null,'%s','%s','%s','%s','%s','%s','%s','%s') %s",\
                                    u.email ,\
                                    u.name  ,\
                                    u.addr  ,\
                                    u.level ,\
                                    u.score ,\
                                    u.phone ,\
                                    u.mobile,\
                                    u.money,
                                    count==9999?"":",");
        strcat(finalsql,values);


        count++;


        if(count == 10000)
        {

//          cout<<finalsql<<endl;
//          break;
           count = 0;
           rc = sqlite3_exec(db, finalsql, nullptr, 0, &zErrMsg);
           if( rc!=SQLITE_OK ){
              fprintf(stderr, "SQL error: %s\n", zErrMsg);
              sqlite3_free(zErrMsg);
           }
           cout<<"1000 pics"<<endl;
        }
    }
    cout<<"game over"<<endl;

    delete []finalsql;
#endif
    return 0;

}
```

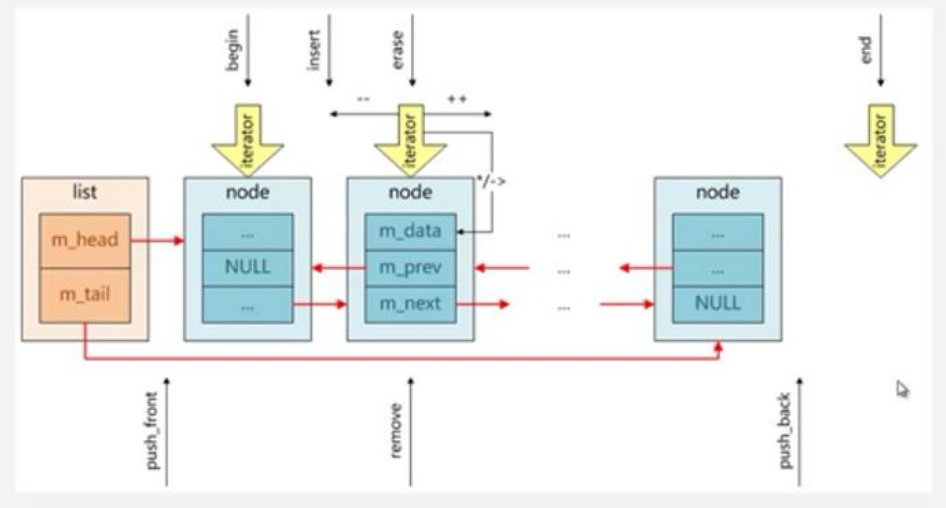
12.3.自实现-模板 List

12.3.1. 引入

```
#include <iostream>
#include <list>

using namespace std;


int main()
{
    list<int> li = {1,3,5,7,9,2,4,6,8,10};
    list<int> li2(li);
    list<int> li3;
    li3 = li;

    li.push_back(19);
    li.push_front(10);
    cout<<"front:"<<li.front()<<endl;
    cout<<"back :"<<li.back()<<endl;
    li.pop_back();
    li.pop_front();
    li.remove(19);

    cout<<"size:"<<li.size()<<endl;
    li.sort([](const int &a, const int & b){return a<b;});

    list<int>::iterator itr;
    for (itr = li.begin();itr != li.end();++itr) {
        cout<<*itr<<endl;
    }
    li.clear();
    cout<<li.empty();
    return 0;
}
```

以上是，我们所知且常用的功能，依据以上功能，我们可以推断出背后的设计思路与常用功能的设计吗？ 以下课程，就是为了，自主设计 List。

12.3.2. 结构



12.3.3. 功能

◆构造，析构，深拷贝
◆首端访问，压入与弹出
◆尾端访问，压入与弹出
◆删除匹配，清空，判空，获取大小
◆输出流重载

12.3.3.1. Node

```
template<typename T>
class Node{
public:
    Node(T const & data, Node *prev=NULL, Node * next=NULL)
        :_data(data),_next(next),_prev(prev){}
    T _data;
    Node *_next;
    Node *_prev;
};
```

12.3.3.2. List

```
template<typename T>
class List{
private:
    Node * _head;
    Node * _tail;
};
```

12.3.4. 声明

```
template<typename T>
class List{
private:
    class Node{
    public:
        Node(T const & data, Node *prev=NULL, Node * next=NULL)
            :_data(data),_next(next),_prev(prev){}
        T _data;
        Node *_next;
        Node *_prev;
    };

public:
    List(void) ;
    ~List(void);
    List(List const & that);
    List & operator=(List const & rhs);
    T & front(void);
    T const & front(void) const;
    void push_front(T const& data);
    void pop_front(void);

    T &back(void);
    T const & back(void)const ;
    void push_back(T const & data);
    void pop_back(void);

    void remove(T const &data);
    void clear(void);
    bool empty(void)const ;
    size_t size(void) const;
    friend ostream& operator<<(ostream& os, List const & list);


private:
    Node * _head;
    Node * _tail;
};
```

12.3.5. 实现

```
#include <iostream>
#include <cstring>
#include <cstdio>

using namespace std;
```

```
template<typename T>

class List{

private:
    class Node{
    public:
        Node(T const & data, Node *prev=nullptr, Node * next=nullptr)
            :_data(data),_next(next),_prev(prev){}
        friend ostream & operator <<(ostream & os, const Node & node){
            os<<node._data;
            return os;
        }
        T _data;
        Node *_next;
        Node *_prev;
    };

public:
    List(void):_head(nullptr),_tail(nullptr) {}
    ~List(void){
        clear();
    }
    List(List const & that)
        :_head(nullptr),_tail(nullptr){
        for(Node *node = that._head; node; node = node->_next)
            push_back(node->_data);
    }
```

12.3.5.1. List & operator=(List const & rhs)

运用临时变量的巧妙思维来实现，赋值重载

```
    List & operator=(List const & rhs){
        if(&rhs != this)
        {
            List list = rhs;
            swap(_head,list._head);
            swap(_tail,list._tail);   //临时对象会消失。
        }
        return *this;
    }
    T & front(void){
        if(empty())
            throw  underflow_error("out front range");
        return _head->_data;
    }
```

12.3.5.2. T const & front(void) const

const_cast< List* >(this)强制转化，是为了避免调用自己，形成自递归，致死循环。

```
T const & front(void) const{
    return const_cast<List*>(this)->front();  //防止出理自递归
}

void push_front(T const& data){
    _head = new Node(data,nullptr,_head);
    if(_head->_next)
        _head->_next->_prev = _head;
    else
        _tail = _head;
}
void pop_front(void){
    if(empty())
        throw  underflow_error("out front range");
    Node *next = _head->_next;
    delete _head;
    _head = next;
    if(_head)
        _head->_prev = nullptr;
    else
        _tail = nullptr;
}

T &back(void){
    if(empty())
        throw  underflow_error("out back range");
    return _tail->_data;
}
T const & back(void)const {
    return const_cast<List*>(this)->back();
}
```

12.3.5.3. void push_back(T const & data)

可以借助 push_back 来，实现拷贝构造。

```
void push_back(T const & data){
    _tail = new Node(data,_tail,nullptr);
    if(_tail->_prev)
        _tail->_prev->_next = _tail;
    else
        _head = _tail;
}
void pop_back(void){
    if(empty())
        throw  underflow_error("out back range");
    Node * prev = _tail->_prev;
```

```
    delete _tail;
    _tail = prev;
    if(_tail)
        _tail->_next = nullptr;
    else {
        _head = nullptr;
    }

}
```

12.3.5.4. void remove(T const &data)

此时要以 node 为中心点，要作处前驱，后继判断。前驱为空，为头，后继为空为尾，单独处理。

还有，相等判断要作特殊化处理，比如 const char *类型的相等判断。

```
void remove(T const &data){
    for (Node *node = _head,*next;    node;   node = next) {
        next = node->_next;
        // if(node->_data == data)
        if(equal(node->_data, data))
        {
            if(node->_prev)   //处理前驱
                node->_prev->_next = node->_next;
            else
                _head = node->_next;

            if(node->_next)   //处理后继
                node->_next->_prev = node->_prev;
            else
                _tail = node->_prev;

            delete node;
        }
    }
}
void clear(void){
    for (Node *next; _head; _head = next) {
        next = _head->_next;
        delete _head;
    }
    _tail = nullptr;
}
bool empty(void)const {
    return !_head && !_tail;
}
size_t size(void) const{
    size_t count =0;
    for(Node * node = _head;node; node = node->_next)
```

```
            ++count;
        return count;
    }
    friend ostream& operator<<(ostream& os, List const & list){
        for(Node *node = list._head; node; node = node->_next)
            os<<*node;
        return os;
    }
private:
    bool equal(const T &a, const T &b) const {
        return a == b;
    }
    Node * _head;
    Node * _tail;
};
template<>
bool List<const char *>::equal(  const char  * const &a, const  char * const &b) const
{
    return  strcmp(a,b) == 0;
}
```

12.3.6. 测试

```
int main()
{
    List<int> Li;
    for(int i=0; i<10; i++)
        Li.push_back(i);

    cout<<Li<<endl;
    List<int> Li2(Li);
    cout<<Li2<<endl;

    List<int> Li3;
    Li3 = Li;
    cout<<Li3<<endl;

    cout<<"size  = "<<Li.size()<<endl;
    cout<<"front = "<<Li.front()<<endl;
    cout<<"back  = "<<Li.back()<<endl;

    Li.pop_back();
    Li.pop_front();
    cout<<Li<<endl;

    for(int i=0; i<10; i++)
        Li.push_back(i);
```

```
    Li.remove(0);
    Li.remove(9);
    Li.remove(5);
    cout<<Li<<endl;

    return 0;
}
```

12.4.自实现-板化迭代器

迭代器是一个类类型的对象，因重载与指针行为一致的运算符(* == != ++)，故其行为与指针类似。

迭代器，可以以类似指针的方式，完全一致透明的访问各类容器，迭代器不为空不为零。

12.4.1. 声明

```
class Iterator{
public:
    Iterator(Node *head= nullptr, Node *tail = nullptr,Node * cur = nullptr);
    bool operator ==(Iterator const &it);
    bool operator !=(Iterator const &it) ;
    bool operator++();
    bool operator++(int);
    bool operator--();
    bool operator--(int);
    T & operator*()const;
    T * operator->() const;
    Iterator begin();
    Iterator end();
private:
    Node * iter_head;
    Node * iter_tail;
    Node * iter_cur;
    friend  class List;
};
```

12.4.2. 实现

13.附录 A

13.1.排序和通用算法(14 个)

函数名	函数功能
inplace_merge	合并两个有序序列，结果序列覆盖两端范围。重载版本使用输入的操作进行排序
	template<class BidIt> void inplace_merge(BidIt first, BidIt middle, BidIt last);
	template<class BidIt, class Pred> void inplace_merge(BidIt first, BidIt middle, BidIt last, Pred pr);
merge	合并两个有序序列，存放到另一个序列。重载版本使用自定义的比较
	template<class InIt1, class InIt2, class OutIt> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt merge(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x, Pred pr);
nth_element	将范围内的序列重新排序，使所有小于第 n 个元素的元素都出现在它前面，而大于它的都出现在后面。重载版本使用自定义的比较操作
	template<class RanIt> void nth_element(RanIt first, RanIt nth, RanIt last);
	template<class RanIt, class Pred> void nth_element(RanIt first, RanIt nth, RanIt last, Pred pr);
partial_sort	对序列做部分排序，被排序元素个数正好可以被放到范围内。重载版本使用自定义的比较操作
	template<class RanIt> void partial_sort(RanIt first, RanIt middle, RanIt last);
	template<class RanIt, class Pred> void partial_sort(RanIt first, RanIt middle, RanIt last, Pred pr);
partial_sort_copy	与 partial_sort 类似，不过将经过排序的序列复制到另一个容器
	template<class InIt, class RanIt> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2);
	template<class InIt, class RanIt, class Pred> RanIt partial_sort_copy(InIt first1, InIt last1,RanIt first2, RanIt last2, Pred pr);
partition	对指定范围内元素重新排序，使用输入的函数，把结果为 true 的元素放在结果为 false 的元素之前
	template<class BidIt, class Pred>

	BidIt partition(BidIt first, BidIt last, Pred pr);
random_shuffle	对指定范围内的元素随机调整次序。重载版本输入一个随机数产生操作
	template<class RanIt> void random_shuffle(RanIt first, RanIt last);
	template<class RanIt, class Fun> void random_shuffle(RanIt first, RanIt last, Fun& f);
reverse	将指定范围内元素重新反序排序
	template<class BidIt> void reverse(BidIt first, BidIt last);
reverse_copy	与 reverse 类似，不过将结果写入另一个容器
	template<class BidIt, class OutIt> OutIt reverse_copy(BidIt first, BidIt last, OutIt x);
rotate	将指定范围内元素移到容器末尾，由 middle 指向的元素成为容器第一个元素
	template<class FwdIt> void rotate(FwdIt first, FwdIt middle, FwdIt last);
rotate_copy	与 rotate 类似，不过将结果写入另一个容器
	template<class FwdIt, class OutIt> OutIt rotate_copy(FwdIt first, FwdIt middle, FwdIt last, OutIt x);
sort	以升序重新排列指定范围内的元素。重载版本使用自定义的比较操作
	template<class RanIt> void sort(RanIt first, RanIt last);
	template<class RanIt, class Pred> void sort(RanIt first, RanIt last, Pred pr);
stable_sort	与 sort 类似，不过保留相等元素之间的顺序关系
	template<class BidIt> void stable_sort(BidIt first, BidIt last);
	template<class BidIt, class Pred> void stable_sort(BidIt first, BidIt last, Pred pr);
stable_partition	与 partition 类似，不过不保证保留容器中的相对顺序
	template<class FwdIt, class Pred> FwdIt stable_partition(FwdIt first, FwdIt last, Pred pr);

13.2.查找算法(13 个)

函数名	函数功能

adjacent_find	在 iterator 对标识元素范围内,查找一对相邻重复元素,找到则返回指向这对元素的第一个元素的 ForwardIterator .否则返回 last.重载版本使用输入的二元操作符代替相等的判断
	template<class FwdIt> FwdIt adjacent_find(FwdIt first, FwdIt last);
	template<class FwdIt, class Pred> FwdIt adjacent_find(FwdIt first, FwdIt last, Pred pr);
binary_search	在有序序列中查找 value,找到返回 true.重载的版本实用指定的比较函数对象或函数指针来判断相等
	template<class FwdIt, class T> bool binary_search(FwdIt first, FwdIt last, const T& val);
	template<class FwdIt, class T, class Pred> bool binary_search(FwdIt first, FwdIt last, const T& val,Pred pr);
count	利用等于操作符,把标志范围内的元素与输入值比较,返回相等元素个数
	template<class InIt, class Dist> size_t count(InIt first, InIt last,const T& val, Dist& n);
count_if	利用输入的操作符,对标志范围内的元素进行操作,返回结果为 true 的个数
	template<class InIt, class Pred, class Dist> size_t count_if(InIt first, InIt last, Pred pr);
equal_range	功能类似 equal,返回一对 iterator,第一个表示 lower_bound,第二个表示 upper_bound
	template<class FwdIt, class T> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val);
	template<class FwdIt, class T, class Pred> pair<FwdIt, FwdIt> equal_range(FwdIt first, FwdIt last,const T& val, Pred pr);
find	利用底层元素的等于操作符,对指定范围内的元素与输入值进行比较.当匹配时,结束搜索,返回该元素的一个 InputIterator
	template<class InIt, class T> InIt find(InIt first, InIt last, const T& val);
find_end	在指定范围内查找"由输入的另外一对 iterator 标志的第二个序列"的最后一次出现.找到则返回最后一对的第一个 ForwardIterator,否则返回输入的"另外一对"的第一个 ForwardIterator.重载版本使用用户输入的操作符代替等于操作
	template<class FwdIt1, class FwdIt2> FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
	template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_first_of	在指定范围内查找"由输入的另外一对 iterator 标志的第二个序列"中任意一个元素的

	第一次出现。重载版本中使用了用户自定义操作符
	template<class FwdIt1, class FwdIt2> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
	template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2, Pred pr);
find_if	使用输入的函数代替等于操作符执行 find
	template<class InIt, class Pred> InIt find_if(InIt first, InIt last, Pred pr);
lower_bound	返回一个 ForwardIterator，指向在有序序列范围内的可以插入指定值而不破坏容器顺序的第一个位置.重载函数使用自定义比较操作
	template<class FwdIt, class T> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val);
	template<class FwdIt, class T, class Pred> FwdIt lower_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
upper_bound	返回一个 ForwardIterator,指向在有序序列范围内插入 value 而不破坏容器顺序的最后一个位置，该位置标志一个大于 value 的值.重载函数使用自定义比较操作
	template<class FwdIt, class T> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val);
	template<class FwdIt, class T, class Pred> FwdIt upper_bound(FwdIt first, FwdIt last, const T& val, Pred pr);
search	给出两个范围，返回一个 ForwardIterator,查找成功指向第一个范围内第一次出现子序列(第二个范围)的位置，查找失败指向 last1,重载版本使用自定义的比较操作
	template<class FwdIt1, class FwdIt2> FwdIt1 search(FwdIt1 first1, FwdIt1 last1,FwdIt2 first2, FwdIt2 last2);
	template<class FwdIt1, class FwdIt2, class Pred> FwdIt1 search(FwdIt1 first1, FwdIt1 last1, FwdIt2 first2, FwdIt2 last2, Pred pr);
search_n	在指定范围内查找 val 出现 n 次的子序列。重载版本使用自定义的比较操作
	template<class FwdIt, class Dist, class T> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val);
	template<class FwdIt, class Dist, class T, class Pred> FwdIt search_n(FwdIt first, FwdIt last,Dist n, const T& val, Pred pr);

13.3.关系算法(8 个)

函数名	函数功能与原型

equal	如果两个序列在标志范围内元素都相等，返回 true。重载版本使用输入的操作符代替默认的等于操作符
	template<class InIt1, class InIt2> bool equal(InIt1 first, InIt1 last, InIt2 x);
	template<class InIt1, class InIt2, class Pred> bool equal(InIt1 first, InIt1 last, InIt2 x, Pred pr);
includes	判断第一个指定范围内的所有元素是否都被第二个范围包含，使用底层元素的<操作符，成功返回 true。重载版本使用用户输入的函数
	template<class InIt1, class InIt2> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
	template<class InIt1, class InIt2, class Pred> bool includes(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
lexicographical_compare	比较两个序列。重载版本使用用户自定义比较操作
	template<class InIt1, class InIt2> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2);
	template<class InIt1, class InIt2, class Pred> bool lexicographical_compare(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, Pred pr);
max	返回两个元素中较大一个。重载版本使用自定义比较操作
	template<class T> const T& max(const T& x, const T& y);
	template<class T, class Pred> const T& max(const T& x, const T& y, Pred pr);
max_element	返回一个 ForwardIterator，指出序列中最大的元素。重载版本使用自定义比较操作
	template<class FwdIt> FwdIt max_element(FwdIt first, FwdIt last);
	template<class FwdIt, class Pred> FwdIt max_element(FwdIt first, FwdIt last, Pred pr);
min	返回两个元素中较小一个。重载版本使用自定义比较操作
	template<class T> const T& min(const T& x, const T& y);
	template<class T, class Pred> const T& min(const T& x, const T& y, Pred pr);

min_element	返回一个 ForwardIterator，指出序列中最小的元素。重载版本使用自定义比较操作
	template<class FwdIt> FwdIt min_element(FwdIt first, FwdIt last);
	template<class FwdIt, class Pred> FwdIt min_element(FwdIt first, FwdIt last, Pred pr);
mismatch	并行比较两个序列，指出第一个不匹配的位置，返回一对 iterator，标志第一个不匹配元素位置。如果都匹配，返回每个容器的 last。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x);
	template<class InIt1, class InIt2, class Pred> pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x, Pred pr);

13.4.删除和替换算法(15 个)

函数名	函数功能
copy	复制序列
	template<class InIt, class OutIt> OutIt copy(InIt first, InIt last, OutIt x);
copy_backward	与 copy 相同，不过元素是以相反顺序被拷贝
	template<class BidIt1, class BidIt2> BidIt2 copy_backward(BidIt1 first, BidIt1 last, BidIt2 x);
iter_swap	交换两个 ForwardIterator 的值
	template<class FwdIt1, class FwdIt2> void iter_swap(FwdIt1 x, FwdIt2 y);
remove	删除指定范围内所有等于指定元素的元素。注意，该函数不是真正删除函数。内置函数不适合使用 remove 和 remove_if 函数
	template<class FwdIt, class T> FwdIt remove(FwdIt first, FwdIt last, const T& val);
remove_copy	将所有不匹配元素复制到一个制定容器，返回 OutputIterator 指向被拷贝的末元素的下一个位置
	template<class InIt, class OutIt, class T> OutIt remove_copy(InIt first, InIt last, OutIt x, const T& val);
remove_if	删除指定范围内输入操作结果为 true 的所有元素
	template<class FwdIt, class Pred> FwdIt remove_if(FwdIt first, FwdIt last, Pred pr);

remove_copy_if	将所有不匹配元素拷贝到一个指定容器
	template<class InIt, class OutIt, class Pred> OutIt remove_copy_if(InIt first, InIt last, OutIt x, Pred pr);
replace	将指定范围内所有等于 vold 的元素都用 vnew 代替
	template<class FwdIt, class T> void replace(FwdIt first, FwdIt last,const T& vold, const T& vnew);
replace_copy	与 replace 类似，不过将结果写入另一个容器
	template<class InIt, class OutIt, class T> OutIt replace_copy(InIt first, InIt last, OutIt x,const T& vold, const T& vnew);
replace_if	将指定范围内所有操作结果为 true 的元素用新值代替
	template<class FwdIt, class Pred, class T> void replace_if(FwdIt first, FwdIt last,Pred pr, const T& val);
replace_copy_if	与 replace_if，不过将结果写入另一个容器
	template<class InIt, class OutIt, class Pred, class T> OutIt replace_copy_if(InIt first, InIt last, OutIt x, Pred pr, const T& val);
swap	交换存储在两个对象中的值
	template<class T> void swap(T& x, T& y);
swap_range	将指定范围内的元素与另一个序列元素值进行交换
	template<class FwdIt1, class FwdIt2> FwdIt2 swap_ranges(FwdIt1 first, FwdIt1 last, FwdIt2 x);
unique	清除序列中重复元素，和 remove 类似，它也不能真正删除元素。重载版本使用自定义比较操作
	template<class FwdIt> FwdIt unique(FwdIt first, FwdIt last);
	template<class FwdIt, class Pred> FwdIt unique(FwdIt first, FwdIt last, Pred pr);
unique_copy	与 unique 类似，不过把结果输出到另一个容器
	template<class InIt, class OutIt> OutIt unique_copy(InIt first, InIt last, OutIt x);
	template<class InIt, class OutIt, class Pred> OutIt unique_copy(InIt first, InIt last, OutIt x, Pred pr);

13.5.生成和变异算法(6 个)

函数名	函数功能
fill	将输入值赋给标志范围内的所有元素

	template<class FwdIt, class T> void fill(FwdIt first, FwdIt last, const T& x);
fill_n	将输入值赋给 first 到 first+n 范围内的所有元素
	template<class OutIt, class Size, class T> void fill_n(OutIt first, Size n, const T& x);
for_each	用指定函数依次对指定范围内所有元素进行迭代访问，返回所指定的函数类型。该函数不得修改序列中的元素
	template<class InIt, class Fun> Fun for_each(InIt first, InIt last, Fun f);
generate	连续调用输入的函数来填充指定的范围
	template<class FwdIt, class Gen> void generate(FwdIt first, FwdIt last, Gen g);
generate_n	与 generate 函数类似，填充从指定 iterator 开始的 n 个元素
	template<class OutIt, class Pred, class Gen> void generate_n(OutIt first, Dist n, Gen g);
transform	将输入的操作作用与指定范围内的每个元素，并产生一个新的序列。重载版本将操作作用在一对元素上，另外一个元素来自输入的另外一个序列。结果输出到指定容器
	template<class InIt, class OutIt, class Unop> OutIt transform(InIt first, InIt last, OutIt x, Unop uop);
	template<class InIt1, class InIt2, class OutIt, class Binop> OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,OutIt x, Binop bop);

13.6.算数算法(4 个)

函数名	函数功能
accumulate	iterator 对标识的序列段元素之和，加到一个由 val 指定的初始值上。重载版本不再做加法，而是传进来的二元操作符被应用到元素上
	template<class InIt, class T> T accumulate(InIt first, InIt last, T val);
	template<class InIt, class T, class Pred> T accumulate(InIt first, InIt last, T val, Pred pr);
partial_sum	创建一个新序列，其中每个元素值代表指定范围内该位置前所有元素之和。重载版本使用自定义操作代替加法
	template<class InIt, class OutIt> OutIt partial_sum(InIt first, InIt last,OutIt result);

	template<class InIt, class OutIt, class Pred> OutIt partial_sum(InIt first, InIt last,OutIt result, Pred pr);
product	对两个序列做内积(对应元素相乘，再求和)并将内积加到一个输入的初始值上。重载版本使用用户定义的操作
	template<class InIt1, class InIt2, class T> T product(InIt1 first1, InIt1 last1,Init2 first2, T val);
	template<class InIt1, class InIt2, class T,class Pred1, class Pred2> T product(InIt1 first1, InIt1 last1,Init2 first2, T val, Pred1 pr1, Pred2 pr2);
adjacent_difference	创建一个新序列，新序列中每个新值代表当前元素与上一个元素的差。重载版本用指定二元操作计算相邻元素的差
	template<class InIt, class OutIt> OutIt adjacent_difference(InIt first, InIt last,OutIt result);
	template<class InIt, class OutIt, class Pred> OutIt adjacent_difference(InIt first, InIt last,OutIt result, Pred pr);

13.7.集合算法(4 个)

函数名	函数功能
set_union	构造一个有序序列，包含两个序列中所有的不重复元素。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_union(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_union(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2,OutIt x, Pred pr);
set_intersection	构造一个有序序列，其中元素在两个序列中都存在。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_intersection(InIt1 first1, InIt1 last1,InIt2 first2,InIt2 last2, OutIt x, Pred pr);
set_difference	构造一个有序序列，该序列仅保留第一个序列中存在的而第二个中不存在的元素。重载版本使用自定义的比较操作
	template<class InIt1, class InIt2, class OutIt> OutIt set_difference(InIt1 first1, InIt1 last1,InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred>

	OutIt set_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);
set_symmetric_difference	构造一个有序序列，该序列取两个序列的对称差集(并集-交集)
	template<class InIt1, class InIt2, class OutIt> OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x);
	template<class InIt1, class InIt2, class OutIt, class Pred> OutIt set_symmetric_difference(InIt1 first1, InIt1 last1, InIt2 first2, InIt2 last2, OutIt x, Pred pr);

13.8.排列组合算法(2 个)

函数名	函数功能
next_permutation	取出当前范围内的排列，并重新排序为下一个排列。重载版本使用自定义的比较操作
	template<class BidIt> bool next_permutation(BidIt first, BidIt last);
	template<class BidIt, class Pred> bool next_permutation(BidIt first, BidIt last, Pred pr);
prev_permutation	取出指定范围内的序列并将它重新排序为上一个序列。如果不存在上一个序列则返回 false。重载版本使用自定义的比较操作
	template<class BidIt> bool prev_permutation(BidIt first, BidIt last);
	template<class BidIt, class Pred> bool prev_permutation(BidIt first, BidIt last, Pred pr);

13.9.堆算法(4 个)

函数名	函数功能
make_heap	把指定范围内的元素生成一个堆。重载版本使用自定义比较操作
	template<class RanIt> void make_heap(RanIt first, RanIt last);
	template<class RanIt, class Pred> void make_heap(RanIt first, RanIt last, Pred pr);
pop_heap	并不真正把最大元素从堆中弹出，而是重新排序堆。它把 first 和 last-1 交换，然后重新生成一个堆。可使用容器的 back 来访问被"弹出"的元素或者使用 pop_back 进行真正的删除。重载版本使用自定义的比较操作

	template<class RanIt> void pop_heap(RanIt first, RanIt last);
	template<class RanIt, class Pred> void pop_heap(RanIt first, RanIt last, Pred pr);
push_heap	假设 first 到 last-1 是一个有效堆，要被加入到堆的元素存放在位置 last-1，重新生成堆。在指向该函数前，必须先把元素插入容器后。重载版本使用指定的比较操作
	template<class RanIt> void push_heap(RanIt first, RanIt last);
	template<class RanIt, class Pred> void push_heap(RanIt first, RanIt last, Pred pr);
sort_heap	对指定范围内的序列重新排序，它假设该序列是个有序堆。重载版本使用自定义比较操作
	template<class RanIt> void sort_heap(RanIt first, RanIt last);
	template<class RanIt, class Pred> void sort_heap(RanIt first, RanIt last, Pred pr);

14.附录 B

dec	ch	dec	ch	dec	ch	dec	ch
0	NUL (空)	32	(空格)	64	@	96	`
1	SOH (标题开始)	33	!	65	A	97	a
2	STX (正文开始)	34	″	66	B	98	b
3	ETX (正文结束)	35	#	67	C	99	c
4	EOT (传送结束)	36	$	68	D	100	d
5	ENQ (询问)	37	%	69	E	101	e
6	ACK (确认)	38	&	70	F	102	f
7	BEL (响铃)	39	’	71	G	103	g
8	BS (退格)	40	(	72	H	104	h
9	HT (横向制表)	41	)	73	I	105	i
10	LF (换行)	42	*	74	J	106	j
11	VT (纵向制表)	43	+	75	K	107	k
12	FF (换页)	44	,	76	L	108	l
13	CR (回车)	45	-	77	M	109	m
14	SO (移出)	46	.	78	N	110	n
15	SI (移入)	47	/	79	O	111	o
16	DLE (退出数据链)	48	0	80	P	112	p
17	DC1 (设备控制 1)	49	1	81	Q	113	q
18	DC2 (设备控制 2)	50	2	82	R	114	r
19	DC3 (设备控制 3)	51	3	83	S	115	s
20	DC4 (设备控制 4)	52	4	84	T	116	t
21	NAK (反确认)	53	5	85	U	117	u
22	SYN (同步空闲)	54	6	86	V	118	v
23	ETB (传输块结束)	55	7	87	W	119	w
24	CAN (取消)	56	8	88	X	120	x
25	EM (媒介结束)	57	9	89	Y	121	y
26	SUB (替换)	58	:	90	Z	122	z
27	ESC (退出)	59	;	91	[	123	{
28	FS (文件分隔符)	60	<	92	\	124	|
29	GS (组分隔符)	61	=	93	]	125	}
30	RS (记录分隔符)	62	>	94	^	126	~
31	US (单元分隔符)	63	?	95	_	127	DEL (删除)