STL学习

1. string 容器

1.1 介绍

string 是**字符数组**，内部维护了一个**动态的字符数组。**

与普通的字符数组相比，string 容器有三个优点：

1. 使用的时候，不用考虑内存分配和释放的问题
2. 动态管理内存（可扩展）
3. 提供了大量操作容器的API

缺点： 效率略有降低

string 类是 std::basic_string 类模板的一个具体化版本的别名。string类是由头文件string提供

```
using std::string = std::basic_string<char,std::char_traits<char>,std::allocator<char>>
```

1.2 构造和析构

静态常量成员 **string::npos** 为字符数组的最大长度（通常为unsigned int）

NBTS（null-terminated string）：C风格的字符串（以空字符0结束的字符串）

1.2.1 string类有七个构造函数（c++ 11 新增了两个）

- **1）string();**

  // 创建一个长度为0的string对象（默认构造函数）。缺省会分配15B的空间（vs2022）

- **2）string(const char* s);**

  // 将string对象初始化为s指向的NBTS（转换函数）。

- 3）**string(const string & str);**

  // 将string对象初始化为str（拷贝构造函数）。

  //! string类中有一个指向动态数组的指针，拷贝构造函数是深拷贝

- 4）**string(const char* s, size_t n);**

  // 将string对象初始化为s指向的地址后n字节的内容。不会判断str的结尾标志

- 5）**string(const string & str, size_t pos = 0, size_t n = npos);**

  // 将sring对象初始化为str从位置pos开始到结尾的字符（或从位置pos开始的n个字符）。会判断str的结尾标志

- 6）**template string(T begin, T end);**

  // 将string对象初始化为区间[begin,end]内的字符，其中begin和end的行为就像指针，用于指定位置，范围包括begin在内，但不包括end。

- //7）**string(size_t n, char c);**

  // 创建一个由n个字符c组成的string对象。

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


int main(void)
{
//  cout << "npos=" << string::npos << endl;

    //TODO: 1) string(); // 创建一个长度为0的string对象（默认构造函数）。

    string s1;
    cout << "s1=" << s1 << endl;
    cout << "s1.capacity()=" << s1.capacity() << endl;  // 返回当前容量，可以存放字符的总数
    cout << "s1.size()=" << s1.size() << endl;  // 返回容器中数据的大小
    cout << "容器动态数组的首地址=" << (void*)s1.c_str() << endl;
    s1 = "xxxxxxxxxxxxxxxxxxxxxxxxxxx";  //! string类扩展容器的时候，先分配更大的空间，然后把内容复制到新的空间，释放掉以前的空间
    cout << "s1.capacity()=" << s1.capacity() << endl;  // 返回当前容量，可以存放字符的总数
    cout << "s1.size()=" << s1.size() << endl;  // 返回容器中数据的大小
    cout << "容器动态数组的首地址=" << (void*)s1.c_str() << endl;
    cout << "-------------------" << endl;

    //TODO: 2) string(const char* s); // 将string对象初始化为s指向的NBTS（转换函数）。
    string s2("hello world");
    cout << "s2=" << s2 << endl;
    string s3 = "hello world";
    cout << "s3=" << s3 << endl;
    cout << "-------------------" << endl;


    //TODO: 3) string(const string & str); // 将string对象初始化为str（拷贝构造函数）。
    //! string类中有一个指向动态数组的指针，拷贝构造函数是深拷贝
    string s4(s3);
    cout << "s4=" << s4 << endl;
    string s5 = s3;
    cout << "s5=" << s5 << endl;
    cout << "-------------------" << endl;


    //TODO: 4) string(const char* s, size_t n);
    // 将string对象初始化为s指向的地址后n字节的内容。不会判断str的结尾标志
    string s6("hello world", 5);
    cout << "s6=" << s6 << endl;
    cout << "s6.capacity()=" << s6.capacity() << endl;
    cout << "s6.size()=" << s6.size() << endl;
    string s7("hello world", 50);
    cout << "s7=" << s7 << endl;
    cout << "s7.capacity()=" << s7.capacity() << endl;
    cout << "s7.size()=" << s7.size() << endl;
    cout << "-------------------" << endl;

    //TODO: 5) string(const string & str, size_t pos = 0, size_t n = npos);
    // 将sring对象初始化为str从位置pos开始到结尾的字符（或从位置pos开始的n个字符）。会判断str的结尾标志
    string s8(s3, 3,5);
    cout << "s8=" << s8 << endl;
    string s9(s3, 3);   // 从s3的第三个位置开始，截取后面所有内容
    cout << "s9=" << s9 << endl;
    cout << "s9.capacity()=" << s9.capacity() << endl;  // 返回当前容量，可以存放字符的总数
    cout << "s9.size()=" << s9.size() << endl;  // 返回容器中数据的大小

    string s10("hello world", 3, 5);
    cout << "s10=" << s10 << endl;
    string s11("hello world", 3);   // 调用的是构造函数 4) string(const char* s, size_t n);
    cout << "s11=" << s11 << endl;
    cout << "-------------------" << endl;


    //TODO: 6) template<class T> string(T begin, T end);
    // 将string对象初始化为区间[begin,end]内的字符，其中begin和end的行为就像指针，用于指定位置，范围包括begin在内，但不包括end。
```

```
    //7) string(size_t n, char c); // 创建一个由n个字符c组成的string对象。
    string s12(8, 'a');
    cout << "s12=" << s12 << endl;
    cout << "s12.capacity()=" << s12.capacity() << endl;
    cout << "s12.size()=" << s12.size() << endl;
    string s13(30, 0);
    cout << "s13=" << s13 << endl;
    cout << "s13.capacity()=" << s13.capacity() << endl;
    cout << "s13.size()=" << s13.size() << endl;

}
```

1.2.2 析构函数~string()释放动态数组的内存空间

1.2.3 C++11新增的构造函数：

- 1）**string(string && str) noexcept;**

  // 将一个string对象初始化为string对象str，并可能修改str（移动构造函数）

- 2） **string(initializer_list il);**

  // 将一个string对象初始化为初始化列表il中的字符

  例如：string ss = {'h','e','l','l','o'};

1.2.4 string容器的设计目标

string容器是以字节为最小存储单元的动态容器。

- 用于存放字符串（不存放空字符0）
- 用于存放数据的内存空间（缓冲区）

string内部有三个指针：

1. **char* start_;　动态分配内存块的首地址**
2. **char* end_;　动态分配内存块最后的地址**
3. **char* finish_;　已使用空间的最后的地址**

string容器有这三个指针，用string存放字符串的时候，不需要空字符0

1.3 特性操作

- **size_t max_size() const;**　// 返回string对象的最大长度string::npos，此函数意义不大。
- **size_t capacity() const;**　// 返回当前容量，可以存放字符的总数。
- **size_t length() const;**　// 返回容器中数据的大小（字符串语义）。
- **size_t size() const;**　// 返回容器中数据的大小（容器语义）。
- **bool empty() const;**　// 判断容器是否为空。
- **void clear();**　// 清空容器。
- **void shrink_to_fit();**　// 将容器的容量降到实际大小（需要重新分配内存）。
- **void reserve(size_t size=0)**　; // 将容器的容量设置为至少size。
- **void resize(size_t len,char c=0)**　; // 把容器的实际大小置为len，如果len<实际大小，会截断多出的部分；如果len>实际大小，就用字符c填充。

1.4 字符操作

- **char &operator ;**
- **const char &operator const;** // 只读。
- **char &at(size_t n);**
- **const char &at(size_t n) const;** // 只读。

operator[]和at()返回容器中的第n个元素，但at函数提供范围检查，当越界时会抛出out_of_range异常，operator[]不提供范围检查。

- **const char *c_str() const;** // 返回容器中动态数组的首地址，语义：寻找以null结尾的字符串。
- **const char *data() const;** // 返回容器中动态数组的首地址，语义：只关心容器中的数据。
- **int copy(char *s, int n, int pos = 0) const;** // 把当前容器中的内容，从pos开始的n个字节拷贝到s中，返回实际拷贝的数目。

1.5 赋值操作

给已存在的容器赋值，将覆盖容器中原有的内容。

- 1） **string &operator=(const string &str);** // 把容器str赋值给当前容器。
- 2） **string &assign(const char *s);** // 将string对象赋值为s指向的NBTS。
- 3） **string &assign(const string &str);** // 将string对象赋值为str。
- 4） **string &assign(const char *s,size_t n);** // 将string对象赋值为s指向的地址后n字节的内容。
- 5） **string &assign(const string &str,size_t pos=0,size_t n=npos);** // 将sring对象赋值为str从位置pos开始到结尾的字符（或从位置pos开始的n个字符）。
- 6） **template string &assign(T begin,T end)** ; // 将string对象赋值为区间[begin,end]内的字符。
- 7） **string &assign(size_t n,char c);** // 将string对象赋值为由n个字符c。

1.6 连接操作

把内容追加到已存在容器的后面。

- 1） **string &operator+=(const string &str);** //把容器str连接到当前容器。
- 2） **string &append(const char *s);** // 把指向s的NBTS连接到当前容器。
- 3） **string &append(const string &str);** // 把容器str连接到当前容器。
- 4） **string &append(const char *s,size_t n);** // 将s指向的地址后n字节的内容连接到当前容器。
- 5） **string &append(const string &str,size_t pos=0,size_t n=npos);** // 将str从位置pos开始到结尾的字符（或从位置pos开始的n个字符）连接到当前容器。
- 6） **template string &append (T begin,T end);** // 将区间[begin,end]内的字符连接到容器。
- 7） **string &append(size_t n,char c);** // 将n个字符c连接到当前容器。

1.7 交换操作

- **void swap(string &str);** // 把当前容器与str交换。

  如果数据量很小，交换的是动态数组中的内容，如果数据量比较大，交换的是动态数组的地址。

1.8 截取操作

- **string substr(size_t pos = 0,size_t n = npos) const;**

  > 返回pos开始的n个字节组成的子容器。

1.9 比较操作

- **bool operator==(const string &str1,const string &str2) const;**

  > 比较两个字符串是否相等。

- **int compare(const string &str) const;**

  > 比较当前字符串和str1的大小。

- **int compare(size_t pos, size_t n,const string &str) const;**

  > 比较当前字符串从pos开始的n个字符组成的字符串与str的大小。

- **int compare(size_t pos, size_t n,const string &str,size_t pos2,size_t n2)const;**

  > 比较当前字符串从pos开始的n个字符组成的字符串与str中pos2开始的n2个字符组成的字符串的大小。

以下几个函数用于和C风格字符串比较。

- **int compare(const char *s) const;**
- **int compare(size_t pos, size_t n,const char *s) const;**
- **int compare(size_t pos, size_t n,const char *s, size_t pos2) const;**

1.10 查找操作

- size_t find(const string& str, size_t pos = 0) const;
- size_t find(const char* s, size_t pos = 0) const;
- size_t find(const char* s, size_t pos, size_t n) const;
- size_t find(char c, size_t pos = 0) const;

- size_t rfind(const string& str, size_t pos = npos) const;
- size_t rfind(const char* s, size_t pos = npos) const;
- size_t rfind(const char* s, size_t pos, size_t n) const;
- size_t rfind(char c, size_t pos = npos) const;

- size_t find_first_of(const string& str, size_t pos = 0) const;

- size_t find_first_of(const char* s, size_t pos = 0) const;
- size_t find_first_of(const char* s, size_t pos, size_t n) const;
- size_t find_first_of(char c, size_t pos = 0) const;

- size_t find_last_of(const string& str, size_t pos = npos) const;
- size_t find_last_of(const char* s, size_t pos = npos) const;
- size_t find_last_of(const char* s, size_t pos, size_t n) const;
- size_t find_last_of(char c, size_t pos = npos) const;

- size_t find_first_not_of(const string& str, size_t pos = 0) const;
- size_t find_first_not_of(const char* s, size_t pos = 0) const;
- size_t find_first_not_of(const char* s, size_t pos, size_t n) const;
- size_t find_first_not_of(char c, size_t pos = 0) const;

- size_t find_last_not_of(const string& str, size_t pos = npos) const;
- size_t find_last_not_of(const char* s, size_t pos = npos) const;
- size_t find_last_not_of(const char* s, size_t pos, size_t n) const;
- size_t find_last_not_of(char c, size_t pos = npos) const;

1.11 替换操作

- string& replace(size_t pos, size_t len, const string& str);
- string& replace(size_t pos, size_t len, const string& str, size_t subpos, size_t sublen = npos);
- string& replace(size_t pos, size_t len, const char* s);
- string& replace(size_t pos, size_t len, const char* s, size_t n);
- string& replace(size_t pos, size_t len, size_t n, char c);

1.12 插入操作

- string& insert(size_t pos, const string& str);
- string& insert(size_t pos, const string& str, size_t subpos, size_t sublen = npos);
- string& insert(size_t pos, const char* s);
- string& insert(size_t pos, const char* s, size_t n);
- string& insert(size_t pos, size_t n, char c);

1.13 删除操作

- **string &erase(size_t pos = 0, size_t n = npos);**

  > 删除pos开始的n个字符。

2. vector容器

vector容器封装了任意类型的动态数组。

包含头文件：#include

vector类模板的声明：

```
// T可以是c++内置的数据类型，结构体和类。第二个模板参数指定分配器，缺省用STL提供的分配器。
template<class T,class Alloc = allocator<T> >
class vector{
private:
    T* start_;  // 动态数组的首地址
    T* finish_; // 动态数组已使用空间的最后地址
    T* end_;    //动态分配内存块最后的地址
    ...
};
```

分配器：

各种STL容器模板都接受一个可选的模板参数，该参数指定使用哪个分配器对象来管理内存。如果省略该模板参数的值，将默认使用allocator，用new和delete分配和释放内存。

2.1 构造与析构

- 1）**vector();**

  > 创建一个空的vector容器。缺省不会分配空间（vs2022）

- 2）**vector(initializer_list il);**

  > 使用统一初始化列表。

- 3）**vector(const vector& v);**

  > 拷贝构造函数。

- 4）**vector(Iterator first, Iterator last);**

  > 用迭代器创建vector容器。（从另一个容器中取一个区间来构造当前容器）

- 5）**vector(vector&& v);**

  > 移动构造函数（C++11标准）。

- 6）**explicit vector(const size_t n);**

  > 创建vector容器，元素个数为n（容量和实际大小都是n）。

- 7）**vector(const size_t n, const T& value);**

  > 创建vector容器，元素个数为n，值均为value。

- **析构函数~vector()释放内存空间。**

2.2 特性操作

- **size_t max_size() const;**

  返回容器的最大长度，此函数意义不大。

- **size_t capacity() const;**

  返回容器的容量。

- **size_t size() const;**

  返回容器的实际大小（已使用的空间）。

- **bool empty() const;**

  判断容器是否为空。

- **void clear();**

  清空容器。

- **void reserve(size_t size);**

  将容器的容量设置为至少size。

- **void shrink_to_fit();**

  将容器的容量降到实际大小（需要重新分配内存）。

- **void resize(size_t size);**

  把容器的实际大小置为size。

- **void resize(size_t size,const T &value);**

  把容器的实际大小置为size，如果size<实际大小，会截断多出的部分；如果size>实际大小，就用value填充。

2.3 元素操作

- **T &operator ;**
- **const T &operator const;** // 只读。
- **T &at(size_t n);**

  访问指定的元素，同时进行越界检查

- **const T &at(size_t n) const;** // 只读。
- **T *data();** // 返回容器中动态数组的首地址。
- **const T *data() const;** // 返回容器中动态数组的首地址。
- **T &front();** // 第一个元素。
- **const T &front();** // 第一个元素，只读。
- **const T &back();** // 最后一个元素，只读。
- **T &back();** // 最后一个元素。

2.4 赋值操作

给已存在的容器赋值，将覆盖容器中原有的内容。

1）**vector &operator=(const vector &v);**

> 把容器v赋值给当前容器。

2）**vector &operator=(initializer_list il);**

> 用统一初始化列表给当前容器赋值。

3）**void assign(initializer_list il);**

> 使用统一初始化列表赋值。

4）**void assign(Iterator first, Iterator last);**

> 用迭代器赋值。

5）**void assign(const size_t n, const T& value);**

> 把n个value给容器赋值。

2.5 交换操作

void swap(vector &v);

> 把当前容器与v交换。交换的是动态数组的地址。

2.6 比较操作

bool operator == (const vector & v) const;

bool operator != (const vector & v) const;

2.7 插入和删除

1）**void push_back(const T& value);**

> 在容器的尾部追加一个元素。

2）**void emplace_back(…);**

> 在容器的尾部追加一个元素，…用于构造元素(...表示可变参数)。C++11

3）**iterator insert(iterator pos, const T& value);**

> 在指定位置插入一个元素，返回指向插入元素的迭代器。

4）**iterator emplace (iterator pos, …);**

> 在指定位置插入一个元素，…用于构造元素，返回指向插入元素的迭代器。C++11

5）**iterator insert(iterator pos, iterator first, iterator last);**

> 在指定位置插入一个区间的元素，返回指向第一个插入元素的迭代器。

```
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };
    vector<int> v1={10,11,12,13,14,15,16,17,18,19,20};
    auto iter=v1.insert(v1.begin()+3, v.begin()+2,v.end()-3);
    cout << "新插入的第一个元素是: " << *iter << endl;

    for (auto it = v1.begin(); it != v1.end(); ++it)
    {
        cout << *it << "  ";
    }
```

6）**void pop_back();**

> 从容器尾部删除一个元素。

7）**iterator erase(iterator pos);**

> 删除指定位置的元素，返回下一个有效的迭代器。

8）**iterator erase(iterator first, iterator last);**

> 删除指定区间的元素，返回下一个有效的迭代器。

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

class AA {
public:
    int m_bh;
    string m_name;

    AA()
    {
        cout << "默认构造函数AA()" << endl;
    }

    AA(const int& bh, const string& name) :m_bh(bh), m_name(name)
    {
        cout << "构造函数, name=" << m_name << endl;
    }

    AA(const AA& gg)
    {
        cout << "拷贝构造函数, name=" << m_name << endl;
    }

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

};
```

```cpp
// main函数，emplace_back参数是c++内置的数据类型，那么用push_back和emplace_bace效果是一样的
    vector<int> v1;
    int a;

    a = 1;      v1.emplace_back(a);
    a = 2;      v1.emplace_back(a);
    a = 3;      v1.emplace_back(a);

    for (int ii = 0; ii < v1.size(); ii++) {
        cout << v1[ii] << "  ";
    }
    cout << endl;
    return 0;
```

```cpp
// main函数，
    vector<AA> v;
    AA a(18, "西施");     // 调用两个参数的构造函数
    v.push_back(a);     // 调用拷贝构造函数

    cout << "bh=" << v[0].m_bh << "  name=" << v[0].m_name << endl;
```

```cpp
//main函数
    vector<AA> v;
    AA a(18, "西施");     // 调用两个参数的构造函数
    v.emplace_back(a);      // 调用拷贝构造函数

    cout << "bh=" << v[0].m_bh << "  name=" << v[0].m_name << endl;
```

```cpp
//main函数
    vector<AA> v;
    v.emplace_back(18, "西施");    // 直接给emplace_back传递参数，调用两个参数的构造函数

    cout << "bh=" << v[0].m_bh << "  name=" << v[0].m_name << endl;
```

2.8 vector的嵌套

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

int main(void)
{
    vector<vector<int>> vv;     // 创建了一个vector容器，元素的数据类型是vector<int>

    vector<int> v;      // 创建了一个容器v,它作为容器vv的元素

    v = { 1,2,3,4,5 };      // 用统一初始化列表给容器v赋值
    vv.push_back(v);        // 将v容器追加追加到vv容器尾部


    v = { 1,2,3,4,5,6,7,8,9,10 };
    vv.push_back(v);

    v = { 21,22,23 };
    vv.push_back(v);

    for (int ii = 0; ii < vv.size(); ii++) {
        for (int jj = 0; jj < vv[ii].size(); jj++) {
            cout << vv[ii][jj] << "  ";
        }
        cout << endl;
```

```
    }
    return 0;
}
```

2.9 注意事项

2.9.1 迭代器失效的问题

resize()、reserve()、assign()、push_back()、pop_back()、insert()、erase()等函数会引起vector容器的动态数组发生变化，可能导致vector迭代器失效。

```
vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };


    for (auto it = v.begin(); it != v.end(); ++it) {
        cout << *it << "  ";
        v.erase(it);
    }
```

- v.erase(it); 从容器中删除迭代器指向的元素，这个迭代器就不存在了，再对它进行++是非法操作内存，程序会崩溃。
- 遍历容器的时候，一般不会进行删除和插入。如果要考虑这种场景，可以这样做：

```
    vector<int> v = { 1,2,3,4,5,6,7,8,9,10 };


    for (auto it = v.begin(); it != v.end();) {
        cout << *it << "  ";
        it=v.erase(it); // erase返回下一个有效的迭代器
    }
```

3. 迭代器

STL把各种数据结构封装成了容器，每个容器都有一个迭代器。

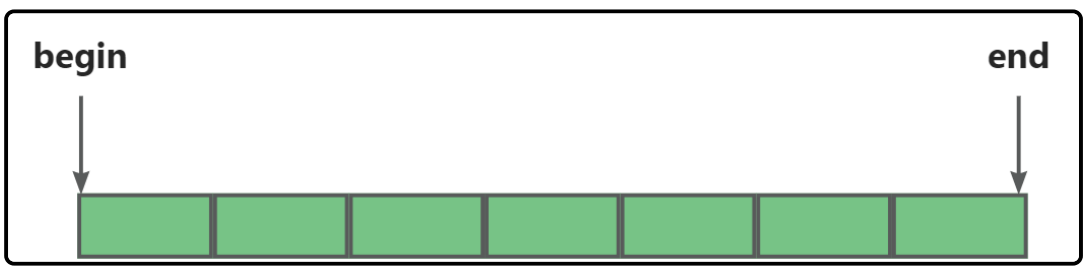
迭代器是访问容器中元素的通用方法。

如果使用迭代器，不同的容器，访问元素的方法是相同的。

迭代器支持的基本操作：赋值（=），解引用（*），比较（==和 !=），从左向右遍历（++） （**只要支持这五个操作就是正向迭代器**）

一般情况下，**迭代器是指针和移动指针的方法。**(相当于位置指针)

- **begin()** 返回容器中第一个元素前面的那个位置
- **end()** 返回容器中最后一个元素后面的那个位置



迭代器有五种分类：

3.1 正向迭代器

支持的操作：赋值（=），解引用（*），比较（==和 !=），从左向右遍历（++）

只能使用++运算符来遍历容器，每次沿容器向右移动一个元素。

- **容器名<元素类型>::iterator 迭代器名;**

  > 正向迭代器。

- **容器名<元素类型>::const_iterator 迭代器名;**

  > 常正向迭代器。

相关的成员函数：

- **iterator begin();**
- **const_iterator begin();**

  > 遍历容器中的元素，但不能通过它修改容器中的元素,只能读取元素的值

- **const_iterator cbegin();**

  > 配合auto使用。eg: auto it2 = vv.cbegin();

- **iterator end();**
- **const_iterator end();**
- **const_iterator cend();**

3.2 双向迭代器

具备正向迭代器的功能，还可以反向（从右往左）遍历容器（也是用++），不管是正向还是反向遍历，都可以用--让迭代器后退一个元素。

支持的操作：赋值（=），解引用（*），比较（==和 !=），从左向右遍历（++） 后退一个元素（--）

- **容器名<元素类型>:: reverse_iterator 迭代器名;**

  > 反向迭代器。

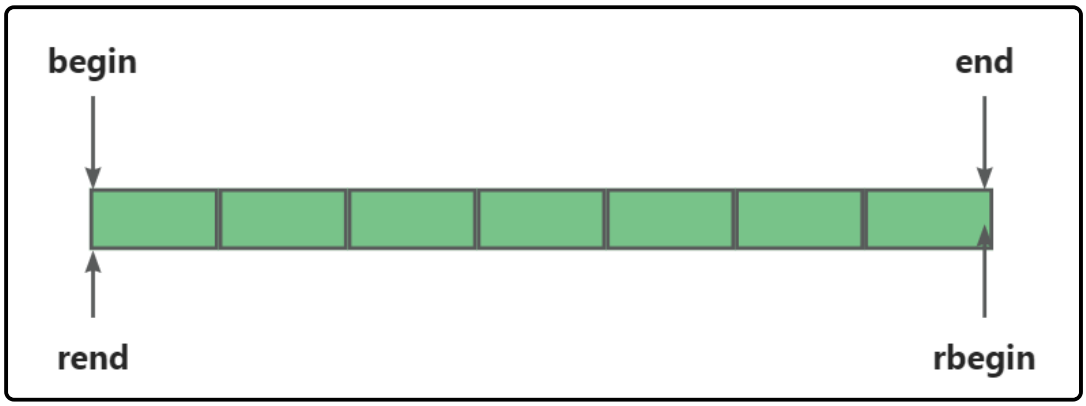
- **容器名<元素类型>:: const_reverse_iterator 迭代器名;**

  > 常反向迭代器。

相关的成员函数：

- **reverse_iterator rbegin();**
- **const_reverse_iterator crbegin();**
- **reverse_iterator rend();**
- **const_reverse_iterator crend();**

> rbegin()　返回容器中最后一个元素后面的那个位置
>
> rend()　　返回容器中第一个元素前面的那个位置



3.3 随机访问迭代器

具备双向迭代器的功能，还支持以下操作：

- 用于比较两个迭代器相对位置的关系运算（<、<=、>、>=）。 (两个迭代器指向的位置，谁在左边，谁在右边)
- **迭代器和一个整数值的加减法运算** （+、+=、-、-=）。
- 支持 **下标运算** （iter[n]）。

数组的指针是纯天然的随机访问迭代器。(数组指针支持迭代器的操作，但不支持迭代器的那些函数begin,end ...)

3.4 输入和输出迭代器

这两种迭代器比较特殊，它们不是把容器当作操作对象，而是把输入/输出流当作操作对象。

4. 基于范围的for循环

对于一个有范围的集合来说，在程序代码中指定循环的范围有时候是多余的，还可能犯错误。

C++ 11 中引入了基于范围的for循环。

语法：

```
for(迭代的变量:迭代的范围)    // 迭代的范围：可以填数组名，容器名，统一初始化列表，其他可迭代的对象
{
    //循环体
}

// 例子：
ector<int> vv = { 1,2,3,4,5,6,7,8,9,10 };
for (auto val : vv) //用基于范围的for循环遍历容器vv,把容器中的元素逐个赋值给val
    {
        cout << val<<"  ";
    }
```

注意：

- 1）迭代的范围可以是数组名、容器名、初始化列表或者可迭代的对象（支持begin()、end()、++、==）。
- 2）数组名传入函数后，已退化成指针，不能作为容器名。
- 3）如果容器中的元素是结构体和类，迭代器变量应该申明为引用，加const约束表示只读。
- 4）注意迭代器失效的问题。

```
vector<int> vv = { 1,2,3,4,5,6,7,8,9,10 };
    for (auto val : vv) //用基于范围的for循环遍历容器vv,把容器中的元素逐个赋值给val
    {
        cout << val<<"  ";
        vv.push_back(10);
    }
结果：迭代器失效
原因：
    在C++中，当你向一个vector中添加元素时（比如使用push_back函数），如果这个操作导致vector的内存空间不足以容纳新元素，那么vector可能会重新分配内存，将元素移动到新的内存位置。这就会导致之前获取的迭代器失效，因为迭代器通常是指向特定内存位置的指针或类似指针的对象。
    在你的代码中，使用基于范围的for循环遍历容器vv，在每次迭代中都向容器中添加一个元素（值为10）。如果在添加元素时，容器的内存空间不足以容纳新元素，vector就会重新分配内存并移动已有的元素到新的内存位置，这会导致之前获取的迭代器失效。因为迭代器仍然指向旧的内存位置，而不是元素移动后的新位置。
```

```
class AA {
public:
    string m_name;
    AA() { cout << "默认构造函数AA()" << endl; }
    AA(const string& name) :m_name{ name } { cout << "构造函数，name=" << m_name << endl; }
    AA(const AA& a) :m_name{ a.m_name } { cout << "拷贝构造函数，name=" << m_name << endl; }
    AA& operator=(const AA& a) { m_name = a.m_name; cout << "赋值函数，name=" << m_name <<
endl; }
    ~AA() { cout << "析构函数，name=" << m_name << endl; }

};

// main函数中
    vector<AA> v;
    cout << "0000,v.capacity()=" << v.capacity() << endl;
    v.emplace_back("西施");
    cout << "1111,v.capacity()=" << v.capacity() << endl;
    v.emplace_back("冰冰");
    // vector容器进行了扩容，扩容分为四个步骤：
    // 1）分配新的可以存放两个元素的内存空间
    // 2）构造新插入的元素冰冰，调用了构造函数AA(const string& name)
    // 3）把西施从原来的内存空间拷过来，调用了拷贝构造函数AA(const AA& a)
    // 4）释放原来的内存空间，调用了析构函数
    cout << "2222,v.capacity()=" << v.capacity() << endl;
    v.emplace_back("蜜蜜");
    cout << "3333,v.capacity()=" << v.capacity() << endl;
```

```
0000,v.capacity()=0
构造函数，name=西施
1111,v.capacity()=1
构造函数，name=冰冰
拷贝构造函数，name=西施
析构函数，name=西施
2222,v.capacity()=2
构造函数，name=蜜蜜
拷贝构造函数，name=西施
拷贝构造函数，name=冰冰
析构函数，name=西施
析构函数，name=冰冰
3333,v.capacity()=3
析构函数，name=西施
析构函数，name=冰冰
析构函数，name=蜜蜜
```

5. list容器

list容器是封装了双链表。支持双向迭代器。

包含头文件：#include

list类的声明：

```
template<class T, class Alloc = allocator<T>>
//T可以是c++内置的数据类型，结构体和类。第二个模板参数指定分配器，缺省用STL提供的分配器
class list{
private:
    iterator head;
    iterator tail;
    ……
}
```

list容器不支持随机访问（不支持+和-运算符）。

5.1 构造函数

1）list();

> 创建一个空的list容器。

2）list(initializer_list il);

> 使用统一初始化列表。

3）list(const list& l);

> 拷贝构造函数。

4）list(Iterator first, Iterator last);

> 用迭代器创建list容器。

5）list(list&& l);

> 移动构造函数（C++11标准）。

6）explicit list(const size_t n);

> 创建list容器，元素个数为n。

7）list(const size_t n, const T& value);

> 创建list容器，元素个数为n，值均为value。

析构函数~list()释放内存空间。

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

int main(void)
{
    list<int> l1;   //创建一个空的list容器
```

```
    cout << "l1.size()=" << l1.size() << endl;
    cout << "-------------------------" << endl;

    list<int> l2{ 1,2,3,4,5,6,7,8,9,10 };       //使用统一初始化列表
    //list<int> l2({ 1,2,3,4,5,6,7,8,9,10 });
    //list<int> l2 = { 1,2,3,4,5,6,7,8,9,10 };
    /*for (auto it = l2.begin(); it != l2.end(); ++it)*/
    for(auto val:l2)
    {
        cout << val << "  ";
    }
    cout <<endl<< "-------------------------" << endl;


    list<int> l3{ l2 };     //使用拷贝构造函数
    for (auto val : l3) {
        cout << val << "  ";
    }
    cout << endl << "-------------------------" << endl;


    list<int> l4(l3.begin(),l3.end());      // 用迭代器创建List容器
    for (auto val : l4) {
        cout << val << "  ";
    }
    cout << endl << "-------------------------" << endl;

    vector<int> v1{ 1,2,3,4,5,6,7,8,9,10 };     // 创建vector容器
    list<int> l5(v1.begin()+2, v1.end()-3);     // 用vector容器的迭代器
    for (auto val : l5) {
        cout << val << "  ";
    }
    cout << endl << "-------------------------" << endl;

    int a1[]= { 1,2,3,4,5,6,7,8,9,10 };
    list<int> l6(a1 + 2, a1 + 10 - 3);  //用数组的指针作为迭代器创建list容器,
    //数组的指针是天然的随机访问迭代器，支持迭代器的操作，但不支持迭代器的那些函数begin,end ...
    for (auto val : l6) {
        cout << val << "  ";
    }
    cout << endl << "-------------------------" << endl;

    char str[] = "hello world";
    string s1(str + 1, str + 7);    // 字符数组的指针，即随机访问迭代器
    for (auto val : s1) {
        cout << val << "  ";
    }
    cout << endl <<s1<<endl<< "-------------------------" << endl;


    vector<int> v2(l3.begin(), l3.end());   //用list容器的迭代器创建vector容器
    for (auto val : v2)
    {
        cout << val << "  ";
    }
```

```
    cout << endl << "------------------------" << endl;
}
```

5.2 特性操作

- **size_t size() const;**

  返回容器的实际大小（已使用的空间）。

- **bool empty() const;**

  判断容器是否为空。

- **void clear();**

  清空容器。

- **void resize(size_t size);**

  把容器的实际大小置为size。

5.3 元素操作

- **T &front();**

  第一个元素。

- **const T &front();**

  第一个元素，只读。

- **const T &back();**

  最后一个元素，只读。

- **T &back();**

  最后一个元素。

5.4 赋值操作

给已存在的容器赋值，将覆盖容器中原有的内容。

- 1）**list &operator=(const list &l);**

  把容器l赋值给当前容器。

- 2）**list &operator=(initializer_list il);**

  用统一初始化列表给当前容器赋值。

- 3）**list assign(initializer_list il);**

  使用统一初始化列表赋值。

- 4）**list assign(Iterator first, Iterator last);**

  用迭代器赋值。

5.5 交换、反转、排序、归并操作

- **void swap(list &l);**

  把当前容器与l交换，交换的是链表结点的地址。

- **void reverse();**

  反转链表。

- **void sort();**

  对容器中的元素进行升序排序。

- **void sort(_Pr2 _Pred);**

  对容器中的元素进行排序，排序的方法由_Pred决定（二元函数）。

- **void merge(list &l);**

  采用归并法合并两个已排序的list容器，合并后的list容器仍是有序

5.6 比较操作

- **bool operator == (const vector & l) const;**
- **bool operator != (const vector & l) const;**

5.7 插入和删除操作

- 1）**void push_back(const T& value);**

  在链表的尾部追加一个元素。

- 2）**void emplace_back(…);**

  在链表的尾部追加一个元素，…用于构造元素。C++11

- 3）**iterator insert(iterator pos, const T& value);**

  在指定位置插入一个元素，返回指向插入元素的迭代器。

- 4）**iterator emplace (iterator pos, …);**

  在指定位置插入一个元素，…用于构造元素，返回指向插入元素的迭代器。C++11

- 5）**iterator insert(iterator pos, iterator first, iterator last);**

  在指定位置插入一个区间的元素，返回指向第一个插入元素的迭代器。

- 6）**void pop_back();**

  从链表尾部删除一个元素。

- 7）**iterator erase(iterator pos);**

  删除指定位置的元素，返回下一个有效的迭代器。

- 8）**iterator erase(iterator first, iterator last);**

  删除指定区间的元素，返回下一个有效的迭代器。

9）**push_front(const T& value);**

> 在链表的头部插入一个元素。

10）**emplace_front(…);**

> 在链表的头部插入一个元素，…用于构造元素。C++11

11）**splice(iterator pos, const vector & l);**

> 把另一个链表连接到当前链表。

12）**splice(iterator pos, const vector & l, iterator first, iterator last);**

> 把另一个链表指定的区间连接到当前链表。

13）**splice(iterator pos, const vector & l, iterator first);**

> 把另一个链表从first开始的结点连接到当前链表。

14）**void remove(const T& value);**

> 删除链表中所有值等于value的元素。

15）**void remove_if(_Pr1 _Pred);**

> 删除链表中满足条件的元素，参数_Pred是一元函数。

16）**void unique();**

> 删除链表中相邻的重复元素，只保留一个。

17）**void pop_front();**

> 从链表头部删除一个元素。

5.8 链表的简单实现

6. pair键值对

pair是类模板，一般用于表示key/value数据，其实是结构体。

> qq号：qq号码就是key　　昵称，等级，生日就是value
>
> 全国人就数据：　省份证号码 是key　　姓名，性别，出生日期 是 value

key的特点：不会重复

value: 可以是一个数据项，也可以是多个数据项，它是除key以外其他数据项的统称。

pair结构模板的定义如下：

```
template <class T1, class T2>
struct pair
{
  T1 first;   // 第一个成员，一般表示key。
  T2 second; // 第二个成员，一般表示value。
```

```
    pair();     // 默认构造函数。

    pair(const T1 &val1,const T2 &val2);  // 有两个参数的构造函数。

    pair(const pair<T1,T2> &p);       // 拷贝构造函数。

    void swap(pair<T1,T2> &p);       // 交换两个pair。

};
```

make_pair函数模板的定义如下：

```
template <class T1, class T2>

pair<T1, T2> make_pair(const T1 &first,const T2 &second)

{

    return pair<T1,T2>(first, second);

}
```

```
    pair<int, string> p0;    // 创建一个空的键值对
    cout << "p0.first=" << p0.first << " ,p0.second=" << p0.second << endl;
    cout << "-----------------------------------------" << endl;

    pair<int, string> p1{ 1,"西施" }; //两个参数的构造函数
    cout << "p1.first=" << p1.first << " ,p1.second=" << p1.second << endl;
    cout << "-----------------------------------------" << endl;


    pair<int, string> p2{ p1 }; //拷贝构造函数
    cout << "p2.first=" << p2.first << " ,p2.second=" << p2.second << endl;
    cout << "-----------------------------------------" << endl;

    auto p3 = pair<int, string>{ 3,"西施" };  //匿名对象,只会调用两个参数的构造函数，不会调用拷贝构造函数
    // 先创建匿名对象，再用了p3的名字。或 创建对象p4时，显示的调用了构造函数
    cout << "p3.first=" << p3.first << " ,p3.second=" << p3.second << endl;
    cout << "-----------------------------------------" << endl;


    auto p5 = make_pair<int,string>(5, "西施");   //  make_pair返回的对象，只会调用两个参数的构造函数，不会调用拷贝构造函数
    cout << "p5.first=" << p5.first << " ,p5.second=" << p5.second << endl;
    cout << "-----------------------------------------" << endl;

    auto p6 = make_pair( 6,"西施" );  //慎用，让make_pair()函数自动推导，再调用拷贝构造(p6=make_pair()返回的键值对)
    cout << "p6.first=" << p6.first << " ,p6.second=" << p6.second << endl;
    cout << "-----------------------------------------" << endl;

    pair<int, string> p7 = make_pair(7,"西施");   //慎用，这里让make_pair()函数自动推导，调用了拷贝构造，再隐式转换
    cout << "p7.first=" << p7.first << " ,p7.second=" << p7.second << endl;
    cout << "-----------------------------------------" << endl;

    p5.swap(p7);
    cout << "p5.first=" << p5.first << " ,p5.second=" << p5.second << endl;
    cout << "p7.first=" << p7.first << " ,p7.second=" << p7.second << endl;

    struct st_girl {
    string name;
    int age;
    double height;
    };
```

```
    // 用pair存放结构体数据
    pair<int, st_girl> p{ 3,{"西施",23,48.6}};
    cout << "p.first=" << p.first << endl;
    cout << "p.second.name=" << p.second.name << endl;
    cout << "p.second.age=" << p.second.age << endl;
    cout << "p.second.height=" << p.second.height << endl;
```

7. 红黑树（平衡二叉排序树）

红黑树的物理结构是二叉链表

```
struct BTNode{
    pair<K,V> p;    // 键值对
    BTNode* parent; // 父节点
    BTNode* lchirld;    // 左子树
    BTNode* rchirld;    // 右子树
};
```

8. map容器

map 容器封装了红黑树（平衡二叉排序树），用于查找。

包含头文件：#include

map容器的元素是 pair键值对。

map类模板的声明：

```
template <class K, class V, class P = less<K>, class _Alloc = allocator<pair<const K, V >>>

class map : public _Tree<_Tmap_traits< K, V, P, _Alloc, false>>

{

   …

}
```

- **第一个模板参数K：key的数据类型（pair.first）。**
- **第二个模板参数V：value的数据类型（pair.second）。**
- **第三个模板参数P：排序方法，缺省按key升序。**
- **第四个模板参数_Alloc：分配器，缺省用new和delete。**

map提供了双向迭代器。

对map容器进行正向的遍历一定会得到一个有序的数列。

二叉链表：

```
struct BTNode
{
    pair<K,V> p;        // 键值对。
    BTNode *parent;    // 父节点。
    BTNode *lchirld;     // 左子树。
    BTNode *rchild;     // 右子树。
};
```

8.1 构造函数

1）**map();**

> 创建一个空的map容器。

2）**map(initializer_list<pair<K,V>> il);**

> 使用统一初始化列表。

3）**map(const map<K,V>& m);**

> 拷贝构造函数。

4）**map(Iterator first, Iterator last);**

> 用迭代器创建map容器。

5）**map(map<K,V>&& m);**

> 移动构造函数（C++11标准）。

```
// 创建一个空的map容器
    map<int, string> m1;

    // 使用统一初始化列表
    map<int, string> m2({{ 8,"冰冰" }, { 3,"西施" }, { 1,"蜜蜜" }, { 7,"金莲" }, { 5,"西瓜" }});
    // 对map容器进行正向的遍历一定会得到一个有序的数列
    for (auto& val : m2) {
        cout << val.first << " ," << val.second << "  ";
    }
    cout << endl;
    cout << "-------------------------------------------------------" << endl;

    // 使用拷贝构造函数
    map<int, string> m3=m2;
    for (auto& val : m3) {
        cout << val.first << " ," << val.second << "  ";
    }
    cout << endl;
    cout << "-------------------------------------------------------" << endl;

    // 用迭代器创建map容器
    auto first = m3.begin();
    first++;
    auto last = m3.end();
    last--;

    map<int, string> m4(first, last);
    for (auto& val : m4) {
        cout << val.first << " ," << val.second << "  ";
    }
    cout << endl;
    cout << "-------------------------------------------------------" << endl;
```

8.2 特性操作

- **size_t size() const;**

  返回容器的实际大小（已使用的空间）。

- **bool empty() const;**

  判断容器是否为空。

- **void clear();**

  清空容器。

8.3 元素操作

K是key的数据类型，V是value的数据类型

- **V &operator ;**

  用给定的key访问元素。

- **const V &operator const;**

  用给定的key访问元素，只读。

- **V &at(K key);**

  用给定的key访问元素。

- **const V &at(K key) const;**

  用给定的key访问元素，只读。

注意：

> 1）[ ]运算符：如果指定键不存在，会向容器中添加新的键值对；如果指定键存在，则读取或修改容器中指定键的值。
>
> 2）at()成员函数：如果指定键不存在，不会向容器中添加新的键值对，而是直接抛出out_of_range 异常。

8.4 赋值操作

给已存在的容器赋值，将覆盖容器中原有的内容。

- 1）**map<K,V> &operator=(const map<K,V>& m);**

  把容器m赋值给当前容器。

- 2）**map<K,V> &operator=(initializer_list<pair<K,V>> il);**

  用统一初始化列表给当前容器赋值。

8.5 交换操作

- **void swap(map<K,V>& m);**

  把当前容器与m交换。交换的是树的根结点。

8.6 比较操作

- **bool operator == (const map<K,V>& m) const;**
- **bool operator != (const map<K,V>& m) const;**

8.7 查找操作

- **查找键值为key的键值对**

  在map容器中查找键值为key的键值对，**如果成功找到，则返回指向该键值对的迭代器；失败返回end()**。

```
iterator find(const K &key);

const_iterator find(const K &key) const;  // 只读。
```

- **查找键值>=key的键值对**

  在map容器中查找第一个键值>=key的键值对，成功返回迭代器；失败返回end()。

```
iterator lower_bound(const K &key);

const_iterator lower_bound(const K &key) const;  // 只读。
```

- **查找键>key的键值对**

  在map容器中查找第一个键值>key的键值对，成功返回迭代器；失败返回end()。

```
iterator upper_bound(const K &key);

const_iterator upper_bound(const K &key) const;  // 只读
```

- **统计键值对的个数**

  统计map容器中键值为key的键值对的个数。

```
size_t count(const K &key) const;
```

8.8 插入和删除

- 1）**void insert(initializer_list<pair<K,V>> il);**

  > 用统一初始化列表在容器中插入多个元素。

- 2）**pair<iterator,bool> insert(const pair<K,V> &value);**

  > 在容器中插入一个元素，返回值pair：first是已插入元素的迭代器，second是插入结果。

- 3）**void insert(iterator first,iterator last);**

  > 用迭代器插入一个区间的元素。

- 4）**pair<iterator,bool> emplace (...);**

  > 将创建新键值对所需的数据作为参数直接传入，map容器将直接构造元素。返回值pair：first是已插入元素的迭代器，second是插入结果。

例：mm.emplace(piecewise_construct, forward_as_tuple(8), forward_as_tuple("冰冰", 18));

第一个参数表示分段构造，第二个参数表示用8构造key，第三个参数表示用"冰冰"和18构造value(超女类)

`piecewise_construct` 用于指示分段构造，`forward_as_tuple(8)` 表示为 `pair` 的 key 构造参数，而 `forward_as_tuple("冰冰", 18)` 表示为 `pair` 的 value 构造参数。

`map` 将会使用给定的参数分别构造 `key` 和 `value` 部分，而不是先构造一个临时的 `pair` 对象再插入到 `map` 中。因此，这行代码只会调用一次 `CGirl` 类的构造函数。

5）**iterator emplace_hint (const_iterator pos,...);**

> 功能与第4）个函数相同，第一个参数提示插入位置，该参数只有参考意义，如果提示的位置是正确的，对性能有提升，如果提示的位置不正确，性能反而略有下降，但是，插入是否成功与该参数元关。该参数常用end()和begin()。成功返回新插入元素的迭代器；如果元素已经存在，则插入失败，返回现有元素的迭代器。

map容器在插入元素之前，会做一次查找的操作，如果我们告诉它插入的位置，他就不用查找了，提高了效率。

6）**size_t erase(const K & key);**

> 从容器中删除指定key的元素，返回已删除元素的个数。

7）**iterator erase(iterator pos);**

> 用迭代器删除元素，返回下一个有效的迭代器。

8）**iterator erase(iterator first,iterator last);**

> 用迭代器删除一个区间的元素，返回下一个有效的迭代器。

```
    map<int, string> m;
    // 使用统一初始化列表在容器中插入多个元素
    m.insert({ {8,"冰冰"},{3,"西施"} });
    m.insert({ pair<int, string>{1, "蜜蜜"}, make_pair<int, string>(7, "金莲"), { 5,"西瓜" }
});
    m.insert({ {18,"冰冰"},{3,"西施1"} });      //插入{3,"西施1"}失败，因为map封装的是红黑树，key
为3已经存在了，所以插入失败

    // 在容器中插入一个元素，返回值pair。first为插入元素的迭代器，second为插入结果
    // pair<iterator,bool> insert(const pair<K,V> value);
    auto ret = m.insert(pair<int, string> {18, "花花"});
    if (ret.second == true) {
        cout << "插入成功"<<"\t";
        cout << ret.first->first << " ," << ret.first->second << endl;
    }
    else {
        cout << "插入失败" << endl;
    }

    // 用迭代器插入一个区间的元素
    // void insert(iterator first,iterator last)
    map<int, string> v{ {2,"张三"},{20,"李四"},{19,"王五"} };
    m.insert(v.begin(), v.end());


    // 将创建新键值对所需的数据作为参数直接传入，map容器将直接构造元素
    // 返回值pair  first是已插入元素的迭代器  second 是插入结果
    // pair<iterator,bool> emplace(...)
    //auto ret1 = m.emplace(pair<int, string>{23, "花花"});
    auto ret1 = m.emplace(23, "花花");
    if (ret1.second == true) {
        cout << "插入成功" << "\t" << ret1.first->first << "  ," << ret1.first->second << endl;
    }
    else {
        cout << "插入失败" << endl;
    }
```

```
    for (auto& val : m) {
        cout << val.first << " ," << val.second << "    ";
    }
    cout << endl;
```

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

class CGirl {
public:
    string m_name;
    int m_age;

    CGirl(const string name, const int age) :m_name{ name }, m_age{ age }
    {
        cout << "两个参数的构造函数" << endl;
    }

    CGirl(const CGirl& g) :m_name{ g.m_name }, m_age{ g.m_age }
    {
        cout << "拷贝构造函数" << endl;
    }
};

int main(void)
{
    map<int, CGirl> mm;
     //mm.insert(pair<int, CGirl>(8, CGirl("冰冰", 18)));  // 一次构造函数，两次拷贝构造函数
    //mm.insert(make_pair<int, CGirl>(8, CGirl("冰冰", 18))); // 一次构造函数，两次拷贝构造函数
    //mm.emplace(pair<int, CGirl>(8, CGirl("冰冰", 18))); // 一次构造函数，两次拷贝构造函数
    //mm.emplace(make_pair<int, CGirl>(8, CGirl("冰冰", 18))); // 一次构造函数，两次拷贝构造函数
    //mm.emplace(8, CGirl("冰冰", 18));    // 一次构造函数，一次拷贝构造函数
    //mm.emplace(8, "冰冰", 18);       // 错误
    mm.emplace(piecewise_construct, forward_as_tuple(8), forward_as_tuple("冰冰", 18)); // 调用了一次构造函数
    // 第一个参数表示分段构造，第二个参数表示用8构造key，第三个参数表示用"冰冰"和18构造value(超女类)

    for (auto& val : mm) {
        cout << val.first << " ," << val.second.m_name << " ," << val.second.m_age << "    ";
    }
    cout << endl;
}
```

mm.insert(make_pair<int, CGirl>(8, CGirl("冰冰", 18))); // 一次构造函数，两次拷贝构造函数

> CGirl("冰冰", 18) 创建了匿名对象，调用了构造函数
>
> 第一次的拷贝构造：构造临时pair对象，会拷贝包含的值，即CGirl对象 ，调用了拷贝构造函数
>
> 第二次的拷贝构造：将构造好的pair对象插入到map中

`emplace` 函数的作用是在 `map` 中插入元素。当你插入一个 `pair` 对象时，它会首先使用你提供的构造参数来构造一个临时的 `pair` 对象，然后将这个临时对象插入到 `map` 中。

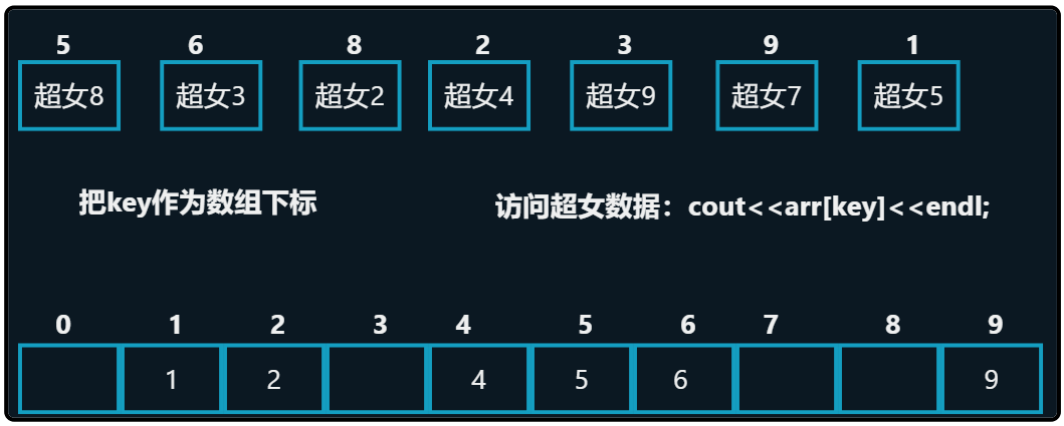
插入一个 `pair` 对象可能会涉及到 `pair` 类型的拷贝构造函数。由于你的 `pair` 包含一个 `CGirl` 对象作为值部分，这可能导致 `CGirl` 类的拷贝构造函数被调用。这是因为在 `pair` 对象的拷贝构造过程中，它会尝试拷贝包含的值，即 `CGirl` 对象。

1. 第一次：构造临时的 `pair` 对象，这个过程中可能会涉及到拷贝构造 `CGirl` 对象。
2. 第二次：将构造好的临时 `pair` 对象插入到 `map` 中，这个过程中也可能会涉及到拷贝构造 `CGirl` 对象。

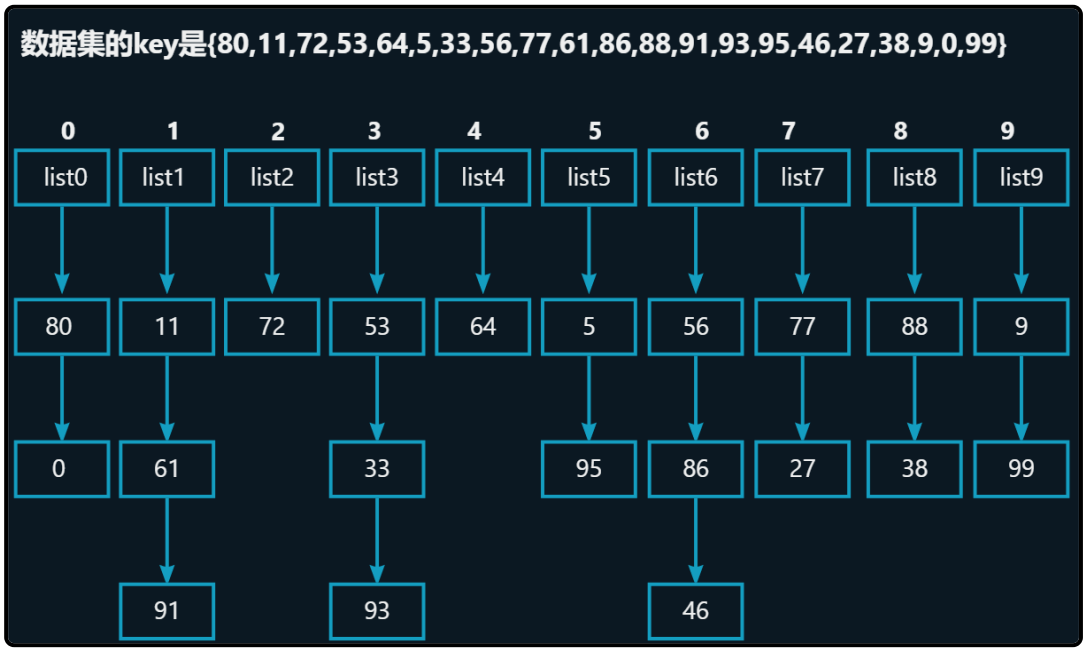
9. 哈希/散列表

哈希表就是数组+链表

1. 第一种：把超女编号作为数组的下标



2. 采用链表数组



> 将数据集%10分成10组，每组的元素用一个链表来存放。（比红黑树的效率高）

哈希表长（桶的个数）：（链表）数组的长度（桶是数组，桶中的元素是链表）

哈希函数：`size_t hash(const T& key) {... // key%小于哈希表长的最大质数}`

装填因子：元素总数/表长，其值越大，效率越低

哈希表的key可以是整数，也可以是字符串。

如果是字符串，怎么返回整数的数组下标？

> 字符串有ascii码,最笨的方法是把它的ascii码加起来，再取余数。

10. unordered_map容器

unorderered_map 容器封装了哈希表，查找，插入和删除元素时，只需要比较几次key的值。

包含头文件： #include <unordered_map>

unordered_map 容器的元素是pair键值对。

在实际开发中，如果数据量只有几万，用红黑树；如果数据量达到千万，必须用哈希表。

unordered_map 类模板的声明：

```
template <class K, class V, class _Hasher = hash<K>, class _Keyeq = equal_to<K>,
    class _Alloc = allocator<pair<const K, V>>>
class unordered_map : public _Hash<_Umap_traits<K, V, _Uhash_compare<K, _Hasher, _Keyeq>,
_Alloc, false>>
{
    …
}
```

第一个模板参数K：key的数据类型（pair.first）。

第二个模板参数V：value的数据类型（pair.second）。

第三个模板参数_Hasher：哈希函数，默认值为std::hash

*第四个模板参数*Keyeq：比较函数，用于判断两个key是否相等，默认值是std::equal_to。

第五个模板参数_Alloc：分配器，缺省用new和delete。

- **创建std::unordered_map类模板的别名** ：

```
template<class K,class V>

using umap = std::unordered_map<K, V>;
```

10.1 构造函数

template<class K,class V>

using umap=std::unordered_map<K,V>; // umap是 unordered_map的别名

因为哈希表自动扩展的代价非常高，所以，构造函数都提供了有桶参数的版本。

1）**umap();**

创建一个空的umap容器。

2）**umap(size_t bucket);**

创建一个空的umap容器，指定了桶的个数(哈希表的长度)，下同。

3）**umap(initializer_list<pair<K,V>> il);**

使用统一初始化列表。

4）**umap(initializer_list<pair<K,V>> il, size_t bucket);**

使用统一初始化列表。

5）**umap(Iterator first, Iterator last);**

用迭代器创建umap容器。

6）**umap(Iterator first, Iterator last, size_t bucket);**

用迭代器创建umap容器。

7）**umap(const umap<K,V>& m);**

拷贝构造函数。

8）**umap(umap<K,V>&& m);**

移动构造函数（C++11标准）。

10.2 特性操作

1）**size_t size() const;**

返回容器中元素的个数。

2）**bool empty() const;**

判断容器是否为空。

3）**void clear();**

清空容器。

4）**size_t max_bucket_count();**

返回容器底层最多可以使用多少桶，无意义。

5）**size_t bucket_count();**

返回容器桶的数量，空容器有8个桶。哈希容器的桶会自动扩展，原理和vector容器相似。

6）**float load_factor();**

返回容器当前的装填因子，load_factor() = size() / bucket_count()。元素的数量/桶数

7）**float max_load_factor();**

返回容器的最大装填因子，当前装填因子达到最大装填因子的值后，容器将扩充，最大装填因子缺省为1。

8）**void max_load_factor (float z);**

设置容器的最大装填因子。

```
unordered_map<int, string> m;
    m.max_load_factor(2);   //设置最大装填因子
    cout << "最大装载因子: " << m.max_load_factor() << endl << endl;;

    m.insert({{1,"西施1"},{2,"西施2"},{3,"西施3"},{4,"西施4"}});
    cout << "当前桶数: " << m.bucket_count() << endl;
    cout << "当前装填因子: " << m.load_factor() << endl;

    cout << "-----------------------------------------------" << endl;

    m.insert({ {5,"西施5"},{6,"西施6"},{7,"西施7"},{8,"西施8"} });
    cout << "当前桶数: " << m.bucket_count() << endl;
    cout << "当前装填因子: " << m.load_factor() << endl;

    cout << "-----------------------------------------------" << endl;

    m.insert({ {15,"西施15"},{16,"西施16"},{17,"西施17"},{18,"西施18"} });
    m.insert({ {19,"西施19"},{20,"西施20"},{21,"西施21"},{22,"西施22"} });
    m.insert({ 9,"西施9" });
```

```
    cout << "当前桶数: " << m.bucket_count() << endl;
    cout << "当前装填因子: " << m.load_factor() << endl;
```

9）**iterator begin(size_t n);**

返回第n个桶中第一个元素的迭代器。

除了普通的迭代器，umap容器的每个桶中还有一个迭代器

10）**iterator end(size_t n);**

返回第n个桶中最后一个元素尾后的迭代器。

```
nordered_map<int, string> m;
m.max_load_factor(5);    //设置最大的装填因子为5

m.insert({ {1,"西施1"},{2,"西施2"},{3,"西施3"},{4,"西施4"} });
m.insert({ {5,"西施5"},{6,"西施6"},{7,"西施7"},{8,"西施8"} });
m.insert({ {15,"西施15"},{16,"西施16"},{17,"西施17"},{18,"西施18"} });
m.insert({ {19,"西施19"},{20,"西施20"},{21,"西施21"},{22,"西施22"} });
m.insert({ 9,"西施9" });

for (auto& val : m) {    //遍历整个容器
    cout << val.first << " ," << val.second << "    ";
}
cout << endl;
cout << "-------------------------------------" << endl;

for (auto it = m.begin(); it != m.end(); it++) {    //遍历整个容器
    cout << it->first << " ," << it->second << "    ";
}
cout << endl;
cout << "-------------------------------------" << endl;

// 遍历全部的桶
for (int ii = 0; ii < m.bucket_count(); ii++) {
    cout << "桶: " << ii << endl;
    for (auto it = m.begin(ii); it != m.end(ii);it++) {
        cout << it->first << " ," << it->second << "    ";
    }
    cout << endl;
}
```

用unordered_map容器可以自动扩容：

umap容器扩容的时候，要把全部已分配的数组和链表销毁，重新哈希（重新分配内存），代价非常大。

在哈希表中，桶是数组，桶中的元素是链表，哈希表要括容，必须重新分配内存，并且因为哈希表的表长不一样，那么哈希函数也肯定不一样，所有的元素要重新哈希，要重新散列。

11）**void reserve(size_t n);**

将容器设置为至少n个桶。

12）**void rehash(size_t n);**

将桶的数量调整为>=n。如果n大于当前容器的桶数，该方法会将容器重新哈希；如果n的值小于当前容器的桶数，该方法可能没有任何作用。

- 13）**size_t bucket_size(size_t n);**

  返回第n个桶中元素的个数，0 <= n < bucket_count()。

- 14）**size_t bucket(K &key);**

  返回值为key的元素对应的桶的编号。

10.3 元素操作

- **V &operator ;**

  用给定的key访问元素。

- **const V &operator const;**

  用给定的key访问元素，只读。

- **V &at(K key);**

  用给定的key访问元素。

- **const V &at(K key) const;**

  用给定的key访问元素，只读。

注意：

1）[ ]运算符：如果指定键不存在，会向容器中添加新的键值对；如果指定键存在，则读取或修改容器中指定键（key）的值。

2）at()成员函数：如果指定键不存在，不会向容器中添加新的键值对，而是直接抛出out_of_range 异常。

10.4 赋值操作

给已存在的容器赋值，将覆盖容器中原有的内容。

- 1）**umap<K,V> &operator=(const umap<K,V>& m);**

  把容器m赋值给当前容器。

- 2）**umap<K,V> &operator=(initializer_list<pair<K,V>> il);**

  用统一初始化列表给容器赋值。

10.5 交换操作

- **void swap(umap<K,V>& m);**

  把当前容器与m交换。

10.6 比较操作

- bool operator == (const umap<K,V>& m) const;
- bool operator != (const umap<K,V>& m) const;

10.7 查找操作

1） **查找键值为key的键值对**

在umap容器中查找键值为key的键值对，如果成功找到，则返回指向该键值对的迭代器；失败返回end()。

iterator find(const K &key);

const_iterator find(const K &key) const; // 只读。

2） **统计键值对的个数**

统计umap容器中键值为key的键值对的个数。

size_t count(const K &key) const;

10.8 插入和删除

1）**void insert(initializer_list<pair<K,V>> il);**

> 用统一初始化列表在容器中插入多个元素。

2）**pair<iterator,bool> insert(const pair<K,V> &value);**

> 在容器中插入一个元素，返回值pair：first是已插入元素的迭代器，second是插入结果。

3）**void insert(iterator first,iterator last);**

> 用迭代器插入一个区间的元素。

4）**pair<iterator,bool> emplace (...);**

> 将创建新键值对所需的数据作为参数直接传入，map容器将直接构造元素。返回值pair：first是已插入元素的迭代器，second是插入结果。

例：mm.emplace(piecewise_construct, forward_as_tuple(8), forward_as_tuple("冰冰", 18));

5）**iterator emplace_hint (const_iterator pos,...);**

> 功能与第4）个函数相同，第一个参数提示插入位置，该参数只有参考意义。对哈希容器来说，此函数意义不大。

6）**size_t erase(const K & key);**

> 从容器中删除指定key的元素，返回已删除元素的个数。

7）**iterator erase(iterator pos);**

> 用迭代器删除元素，返回下一个有效的迭代器。

8）**iterator erase(iterator first,iterator last);**

> 用迭代器删除一个区间的元素，返回下一个有效的迭代器。

11. queue容器

queue容器的逻辑结构是队列，物理结构可以数组或链表，主要用于多线程之间的数据共享。

包含头文件：#include

queue类模板的声明：

```
template <class T, class _Container = deque<T>>

class queue{

    ……

}
```

- 第一个模板参数T：元素的数据类型。
- 第二个模板参数_Container：底层容器的类型， **缺省是std::deque** ，可以用 **std::list** ，还可以用 **自定义的类模板。** vector容器不支持队列。
- **queue容器不支持迭代器。** queue容器把其他容器做了二次封装，

11.1 构造函数

- 1）**queue();**

  创建一个空的队列。

  创建队列的时候，如果把第二个模板参数指定为list,那么队列中就有一个list容器的对象。

- 2）**queue(const queue& q);**

  拷贝构造函数。

- 3）**queue(queue&& q);**

  移动构造函数（C++11标准）。

- 析构函数~queue()释放内存空间。

11.2 常用操作

- 1）**void push(const T& value);**

  元素入队。

- 2）**void emplace(…);**

  元素入队，…用于构造元素。C++11

- 3）**size_t size() const;**

返回队列中元素的个数。

4）**bool empty() const;**

判断队列是否为空。

5）**T &front();**

返回队头元素。

6）**const T &front();**

返回队头元素，只读。

7）**T &back();**

返回队尾元素。

8）**const T &back();**

返回队头元素，只读。

9）**void pop();**

出队，删除队头的元素。

11.3 其他操作

1）**queue &operator=(const queue &q);**

赋值。

2）**void swap(queue &q);**

交换。

3）**bool operator == (const queue & q) const;**

重载==操作符。

4）**bool operator != (const queue & q) const;**

重载!=操作符。

12. STL其他容器

STL的其他容器都是在堆上分配内存，只有array在栈上分配内存。

12.1 array(静态数组)

12.1.1 物理结构

在栈上分配内存，创建数组的时候，数组长度必须是常量，创建后的数组大小不可变。

```
template<class T, size_t size>
class array{
private:
```

T elems_[size];

.....

};

12.1.2 迭代器

随机访问迭代器

12.1.3 特点

部分场景中，比常规数组更方便（能用于模板），可以代替常规数组。

12.1.4 各种操作

- 1）**void fill(const T & val);**

  给数组填充值（清零）。

- 2）**size_t size();**

  返回数组的大小。

- 3）**bool empty() const;**

  无意义。

- 4）**T &operator ;**

- 5）**const T &operator const;** // 只读。

- 6）**T &at(size_t n);**

- 7）**const T &at(size_t n) const;** // 只读。

- 8）**T *data();**

  // 返回数组的首地址。

- 9）**const T *data() const;**

  返回数组的首地址。

- 10）**T &front();** // 第一个元素。

- 11）**const T &front();** // 第一个元素，只读。

- 12）**const T &back();** // 最后一个元素，只读。

- 13）**T &back();** // 最后一个元素。

```
//int aa[11] = { 1,2,3,4,5,6,7,8,9,10 };
array<int,10>aa= { 1,2,3,4,5,6,7,8,9,10 };  // 一维数组

for (int ii = 0; ii < 10; ii++) {   // 传统的方法
    cout << aa[ii] << " ";
}
cout << endl;

for (auto it = aa.begin(); it != aa.end(); it++) {  // 使用迭代器
    cout << *it << " ";
}
cout << endl;

for (int ii = 0; ii < aa.size(); ii++) {    // 利用array的size的方法
    cout << aa[ii] << " ";
```

```
    }
    cout << endl;

    for (auto val : aa) {    //基于范围的for循环
        cout << val << " ";
    }
    cout << endl;
```

```
//普通的二维数组

//void func(int arr[][5], int len)
void func(int (*arr)[6],int len)
{
    for (int ii = 0; ii < 10; ii++) {
        for (int jj = 0; jj < 6; jj++) {
            cout << arr[ii][jj] << " ";
        }
        cout << endl;
    }
}


    int bb[10][6];
    for (int ii = 0; ii < 10; ii++) {
        for (int jj = 0; jj < 6; jj++) {
            bb[ii][jj] = jj * 10 + ii;
        }
    }

    func(bb, 10);
```

```
// array容器的二维数组

template <typename T>
void func(const T& arr)
{
    for (int ii = 0; ii < arr.size(); ii++) {
        for (int jj = 0; jj < arr[ii].size(); jj++) {
            cout << arr[ii][jj] << " ";
        }
        cout << endl;
    }
}

array<array<int, 6>, 10> bb; // 二维数组，相当于 int bb[10][5]
    for (int ii = 0; ii < bb.size(); ii++) {
        for (int jj = 0; jj < bb[ii].size(); jj++) {
            bb[ii][jj] = jj * 10 + ii;
        }
    }

    func(bb);
```

12.2 deque(双端队列)

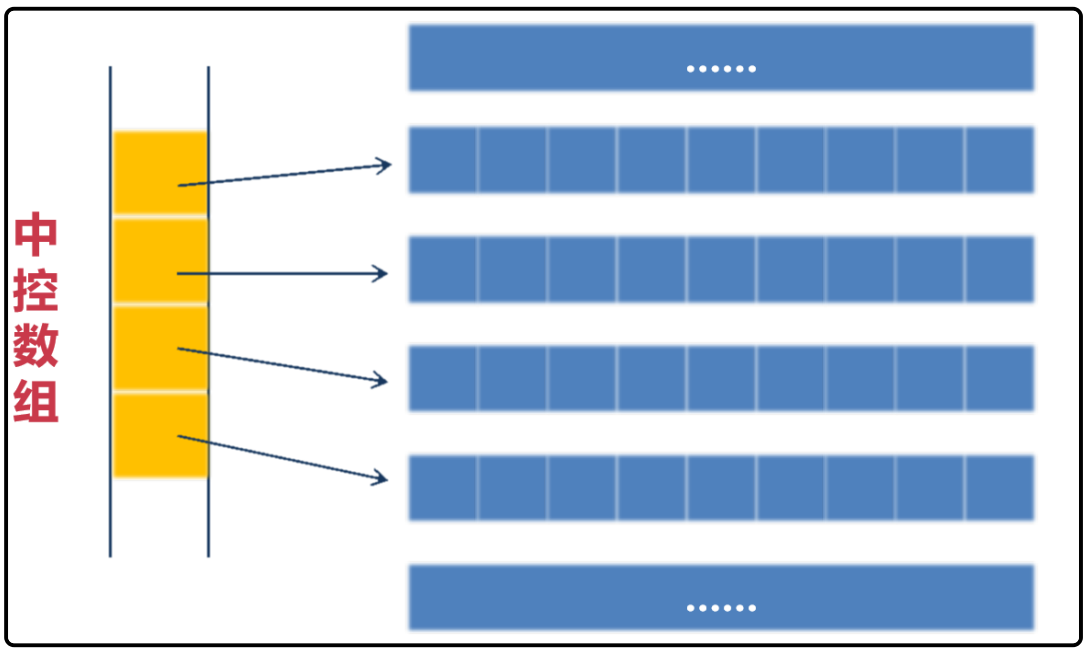
deque容器用来做队列和栈的底层容器比较好。

12.2.1物理结构

deque容器存储数据的空间是多段等长的连续空间构成，各段空间之间并不一定是连续的。

为了管理这些连续空间的分段，deque容器用一个数组存放着各分段的首地址。

deque容器的存储空间由多块内存组成，每块内存的大小是一样的，一块内存可以存放多个元素。即 一段一段的分配内存。



通过建立数组，deque容器的分段的连续空间能实现整体连续的效果。

当deque容器在头部或尾部增加元素时，会申请一段新的连续空间，同时在数组中添加指向该空间的指针。

12.2.2 迭代器

随机访问迭代器

12.2.3 特点

- 提高了在 **两端** 插入和删除元素的效率，扩展空间的时候，不需要拷贝以前的元素。
- 在中间插入和删除元素的效率比vector更糟糕。
- 随机访问的效率比vector容器略低。

12.2.4 各种操作

与vector容器相同。

12.3 forward_list (单链表)

12.3.1 物理结构

单链表

12.3.2 迭代器

正向迭代器

12.3.3 特点

比双链表少了一个指针，可节省一丢丢内存，减少了两次对指针的赋值操作。

如果单链表能满足业务需求，建议使用单链表而不是双链表。

12.3.4 各种操作

与list容器相同

12.4 multimap

底层是红黑树。

multimap与map的区别：multimap允许关键字（key）重复，而map不允许重复

各种操作与map容器相同。

12.5 set&multiset

底层是红黑树。

set和map的区别：map中存储的是pair键值对，而set只保存关键字。

multset与set的区别：multiset允许关键字（key）重复，而set不允许重复。

各种操作与map容器相同。

12.6 unordered_multimap

底层是哈希表。

unordered_map 与 unordered_multimap区别：unordered_multimap允许关键字重复，unordered_map不允许关键字重复。

各种操作与unordered_map相同。

12.7 unordered_set & unordered_multiset

底层是哈希表。

unordered_set 与 unordered_multiset 的区别：unordered_set 不允许关键字重复，unordered_multiset 允许关键字重复。

unordered_set 与 unordered_map 的区别：unordered_map存储的是键值对，unordered_set 只保存关键字。

各种操作与unordered_map容器相同。

12.8 priority_queue (优先队列)

优先级队列相当于一个有权值的单向队列queue，在这个队列中，所有元素是按照优先级排列的。

底层容器可以用deque和list。

各种操作与queue容器相同。

13. STL算法

STL提供了很多处理容器的函数模板，它们的设计是相同的，有以下特点：

1）用迭代器表示需要处理数据的区间。

2）返回迭代器放置处理数据的结果

3）接受一个函数对象参数（结构体模板），用于数据处理。

13.1 函数对象

很多STL算法都是用函数对象，也叫函数符（functor）,包含函数名，函数指针和仿函数。 **仿函数的本质是类，调用仿函数就是调用类中的成员函数。仿函数可以使用成员变量来传递参数，更方便。（c++建议用仿函数）**

函数符的概念：

1）生成器（generator）：不用参数就可以调用的函数符

2）一元函数（unary function）：用一个参数可以调用的函数符

3）二元函数（binary function）：用两个参数可以调用的函数符

改进的概念：

1）一元谓词（predicate）：返回bool值的一元函数。

2）二元谓词（binary predicate）：返回bool值的二元函数。

13.2 预定义的函数对象

STL定义了多个基本的函数符（函数对象），用于支持STL的算法函数。

包含头文件：**#include**

运算符和相应的函数符（函数对象）：

运算符	函数符
+	plus
–	minus
*	multiplies
/	divides
%	modulus
–	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal

运算符	函数符
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

```
template <class _Ty = void>
struct greater {
    _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef _Ty _FIRST_ARGUMENT_TYPE_NAME;
    _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef _Ty _SECOND_ARGUMENT_TYPE_NAME;
    _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS typedef bool _RESULT_TYPE_NAME;

    _NODISCARD constexpr bool operator()(const _Ty& _Left, const _Ty& _Right) const {
        return _Left > _Right;
    }
};
```

13.3 算法函数

STL将算法函数分成四组：

1）非修改式序列操作：对区间中的每个元素进行操作，这些操作不修改容器的内容。

2）修改式序列操作：对区间中的每个元素进行操作，这些操作可以容器的内容（可以修改值，也可以修改排列顺序）。

3）排序和相关操作：包括多个排序函数和其它各种函数，如集合操作。

4）通用数字运算：包括将区间的内容累积、计算两个容器的内部乘积、计算小计、计算相邻对象差的函数。通常，这些都是数组的操作特性，因此　　vector是最有可能使用这些操作的容器。

前三组在头文件#include 中，第四组专用于数值数据，在#include 中。

非修改式序列操作

函　数	描　述
all_of( )	如果对于所有元素的谓词测试都为 true，则返回 true。这是 C++11 新增的
any_of( )	只要对于任何一个元素的谓词测试为 true，就返回 true。这是 C++11 新增的
none_of( )	如果对于所有元素的谓词测试都为 false，则返回 true。这是 C++11 新增的
for_each( )	将一个非修改式函数对象用于区间中的每个成员
find( )	在区间中查找某个值首次出现的位置
find_if( )	在区间中查找第一个满足谓词测试条件的值
find_if_not( )	在区间中查找第一个不满足谓词测试条件的值。这是 C++11 新增的
find_end( )	在序列中查找最后一个与另一个序列匹配的值。匹配时可以使用等式或二元谓词
find_first_of( )	在第二个序列中查找第一个与第一个序列的值匹配的元素。匹配时可以使用等式或二元谓词
adjacent_find	查找第一个与其后面的元素匹配的元素。匹配时可以使用等式或二元谓词
count( )	返回特定值在区间中出现的次数
count_if( )	返回特定值与区间中的值匹配的次数，匹配时使用二元谓词
mismatch( )	查找区间中第一个与另一个区间中对应元素不匹配的元素，并返回指向这两个元素的迭代器。匹配时可以使用等式或二元谓词
equal( )	如果一个区间中的每个元素都与另一个区间中的相应元素匹配，则返回 true。匹配时可以使用等式或二元谓词
is_permutation( )	如果可通过重新排列第二个区间，使得第一个区间和第二个区间对应的元素都匹配，则返回 true，否则返回 false。匹配可以是相等，也可以使用二元谓词进行判断。这是 C++11 新增的
search( )	在序列中查找第一个与另一个序列的值匹配的值。匹配时可以使用等式或二元谓词
search_n( )	查找第一个由 n 个元素组成的序列，其中每个元素都与给定值匹配。匹配时可以使用等式或二元谓词

1. all_of( )（C++11）

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 true，或者该区间为空，则函数 all_of( )返回 true；否则返回 false。

2. any_of( )（C++11）

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last,
            Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 any_of( )返回 false；否则返回 true。

3. none_of( )（C++11）

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last,
             Predicate pred);
```

如果对于区间[first, last]中的每个迭代器，pred(*i)都为 false，或者该区间为空，则函数 all_of( )返回 true；否则返回 false。

4. for_each( )

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last,
                  Function f);
```

for_each( )函数将函数对象 f 用于[first, last]区间中的每个元素，它也返回 f。

5. find( )

```
template<class InputIterator, class T
InputIterator find(InputIterator first, InputIterator last,
                   Const T& value);
```

find( )函数返回一个迭代器，该迭代器指向区间[first, last]中第一个值为 value 的元素；如果没有找到这样的元素，则返回 last。

6. find_if( )

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if( )函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 true 的元素；如果没有找到这样的元素，则返回 last。

7. find_if_not( )

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                      Predicate pred);
```

find_if_not( )函数返一个迭代器，该迭代器指向[first, last]区间中第一个对其调用函数对象 pred(*i)时结果为 false 的元素；如果没有找到这样的元素，则返回 last。

8. find_end( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
    class BinaryPredicate>
ForwardIterator1 find_end(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_end( )函数返回一个迭代器，该迭代器指向[first1, last1] 区间中最后一个与[first2, last2] 区间的内容匹配的序列的第一个元素。第一个版本使用值类型的= =运算符来比较元素；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都返回 last1。

9. find_first_of( )

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2);

template<class ForwardIterator1, class ForwardIterator2,
        class BinaryPredicate>
ForwardIterator1 find_first_of(
        ForwardIterator1 first1, ForwardIterator1 last1,
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

find_first_of( ) 函数返回一个迭代器，该迭代器指向区间[first1, last1]中第一个与[first2, last2]区间中的任何元素匹配的元素。第一个版本使用值类型的==运算符对元素进行比较；第二个版本使用二元谓词函数对象 pred 来比较元素。也就是说，如果 pred(*it1, *it2)为 true，则 it1 和 it2 指向的元素匹配。如果没有找到这样的元素，则它们都将返回 last1。

13.4 学习要领

1）如果容器有成员函数，就是用成员函数；如果没有则考虑使用STL的算法函数。

2）如果要采用某算法函数，一定要搞清楚它的原理，关注它的效率。

13.5 for_each() 遍历

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

// 传统的思想是 传对象的地址或引用，只能满足某种类型的容器
// 模板的思想是 传迭代器，函数可以支持多种容器，只要它有迭代器就行了

void foreach(const vector<int>& v)
{
    for (auto& val : v) {
        cout << "亲爱的" << val << "号，我是一只傻傻鸟" << endl;
    }
}


template <typename T>
void foreach(const T first, const T last)
{
    for (auto it = first; it != last; ++it) {
        cout << "亲爱的" << *it << "号，我是一只傻傻鸟" << endl;;
    }
}

//void foreach(const vector<string>::iterator first, const vector<string>::iterator last)
//{
//  for (auto it = first; it != last; ++it) {
//      cout << "亲爱的" << *it << "号，我是一只傻傻鸟" << endl;;
//  }
//}

int main(void)
{
    //vector<int> bh = { 5,8,2,6,9,3,1,7 };
    //vector<string> bh = { "05","08","02","06","09","03","01","07" };
    list<string> bh = { "05","08","02","06","09","03","01","07" };  //list容器不支持随机访问
    //foreach(bh);  //不支持向部分超女表白

    foreach(bh.begin(), bh.end()); //支持vector容器的元素的类型为其他数据类型，支持其他容器

    return 0;
}
```

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

// 传统的思想是 传对象的地址或引用，只能满足某种类型的容器
// 模板的思想是 传迭代器，函数可以支持多种容器，只要它有迭代器就行了

template <typename T>
void love(const T& str)
{
    cout << "亲爱的" << str << "号，我是一只傻傻鸟" << endl;;
}

template <class T>
class czs {
public:
    void operator()(const T& str)
    {
        cout << "亲爱的" << str << "号，我是一只傻傻鸟" << endl;;
    }

};

//template <typename T>
//void foreach(const T first, const T last,void (*pfunc)(const string&))
//{
//  for (auto it = first; it != last; ++it) {
//      pfunc(*it); // 以超女对象编号为实参回调用户自定义的函数
//  }
//}
//
//
//template <typename T>
//void foreach(const T first, const T last, czs& pfunc)
//{
//  for (auto it = first; it != last; ++it) {
//      pfunc(*it); // 以超女对象为实参调用类的operator()函数
//  }
//}

// 将上面两个模板合并成一个模板
template <typename T1,typename T2>
void foreach(const T1 first, const T1 last, T2 pfunc)
{
    for (auto it = first; it != last; ++it) {
        pfunc(*it);
    }
}

//void foreach(const vector<string>::iterator first, const vector<string>::iterator last)
//{
//  for (auto it = first; it != last; ++it) {
//      cout << "亲爱的" << *it << "号，我是一只傻傻鸟" << endl;;
//  }
//}

int main(void)
{
    vector<int> bh = { 5,8,2,6,9,3,1,7 };
    //vector<string> bh = { "05","08","02","06","09","03","01","07" };
    //list<string> bh = { "05","08","02","06","09","03","01","07" };    //list容器不支持随机访问

    foreach(bh.begin(), bh.end(), czs<int>()); //支持vector容器的元素的类型为其他数据类型，支持其他容器

    foreach(bh.begin(), bh.end(),love<int> );

    return 0;
}
```

上面的代码实现了for_each()函数。

for_each()的声明：

template<class InputIterator,class Function>

Function for_each(InputIterator first, InputIterator last , Function f);

for_each()函数将函数对象f用于[first,last]区间中的每个元素，**它也返回f。**

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

template <class T>
class girl {
public:
    T m_yz;
    int m_count;    // 符合条件的元素个数
    girl(const T yz):m_yz(yz),m_count(0) { }

    void operator()(const T& yz) {
        if (yz == m_yz) {
            m_count++;
        }
    }
};


int main(void)
{
    vector<int> vv = { 1,3,2,4,1,2,3,1,4,3 };

    girl<int> g = for_each(vv.begin(),vv.end(),girl<int>(1));    // 按颜值统计超女个数
    // 创建匿名对象，把yz传进去，m_yz=1 ，统计m_yz=1的个数，最后返回对象

    cout << "g.m_count=" << g.m_count << endl;
}
```

仿函数不仅可以携带参数，还可以保存结果。

13.6 find_if() 查找

原理：遍历容器中的元素，在遍历的过程中，把元素的值作为参数传递给函数；如果函数的返回值为true,那就不再遍历了，find_if()函数返回刚才这个　　元素的迭代器；在遍历的过程中，如果返回的是false，那么find_if()函数返回最后一个元素的后面（返回end()迭代器）。

声明：

template<class InputIterator, class Predicate>

InputIterator find_if(InputIterator first, InputIterator last , Predicate pred);

// 第一个(first)，第二个参数(last)都是迭代器，第三个参数是函数对象

注意：全部的STL函数的前两个参数都是迭代器，用于指示容器中元素的区间.

函数实现：

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

template <typename T>
bool love(const T& str)
{
```

```
    if (str != 3) return false;
    cout << "亲爱的" << str << "号，我是一只傻傻鸟" << endl;
    return true;
}

template <class T>
class zxc {
public:
    T m_no;
    zxc(const T& no):m_no(no) { }
    bool operator()(const T& str)
    {
        if (str != m_no) return false;
        cout << "亲爱的" << str << "号，我是一只傻傻鸟" << endl;
        return true;
    }
};

template <typename T1,typename T2>
T1 findif(const T1 first, const T1 last, T2 pfunc)
{
    for (auto it = first; it != last; ++it) {
        if (pfunc(*it) == true) {
            return it;
        }
    }
    return last;
}

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

    auto it1=find_if(bh.begin(),bh.end(),love<int>);
    if (it1 == bh.end()) {
        cout << "查找失败" << endl;
    }
    else {
        cout << "查找成功: " <<*it1<< endl;
    }

    auto it2 = find_if(bh.begin(), bh.end(), zxc<int>(8));
    if (it2 == bh.end()) {
        cout << "查找失败" << endl;
    }
    else {
        cout << "查找成功: " << *it2 << endl;
    }
    return 0;
}
```

注意：

算法回调的函数可以是普通函数，也可以是仿函数。

仿函数的本质是类，调用仿函数就是调用类中的成员函数。仿函数可以使用成员变量来传递参数，更方便。

13.7 冒泡排序（bsort）

代码实现：

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>    // STL算法函数
#include <functional>   // STL仿函数
using namespace std;

template <typename T>
bool compasc(const T& left, const T& right) // 用于升序
```

```
{
    /*if (left < right) return true;
    return false;*/
    return left < right;
}

template <class T>
class _creat {
public:
    bool operator()(const T& left, const T& right) {    //仿函数，升序
        return left < right;
    }
};


template <typename T>
bool compdesc(const T& left, const T& right)    // 用于降序
{
    /*if (left > right) return true;
    return false;*/
    return left>right;
}

template <class T>
class _less {
public:
    bool operator()(const T& left, const T& right) {    //仿函数，降序
        return left > right;
    }
};

template <typename T1,typename compare>
void bsort(const T1 first, const T1 last,compare comp)
{
    while (true)
    {
        bool bswap = false; // 本轮遍历已交换过元素的标识，true 交换过  false 未交换过
        for (auto it = first; ;) {
            auto left = it;     // 左边的元素
            it++;
            auto right = it;    // 右边的元素
            if (right == last) {        // 表示it已经是最后一个元素了
                break;
            }

            // 如果是c++内置的数据类型或字符串，可以用>和< ；如果是自定义的数据类型，就不能用 < 和 >
了
            //! 要满足用户需求，用 回调程序员自定义的函数
            //if (*left < *right) { //如果左边的元素比右边大，交换他们的值,升序

            // 如果comp()返回true,left排在前面，否则right排在后面
            if (comp(*left, *right) == true) {
                continue;
            }

            auto tmp = *left;
            *left = *right;
            *right = tmp;
            bswap = true;   //一轮遍历已交换元素的标识
        }

        if (bswap == false) {   // 如果在for循环中不曾交换过元素，说明全部的元素已有序
            break;
        }
    }

}


int main(void)
{
    vector<int> bh = { 5,8,2,6,9,33,1,7 };
    //vector<string> bh = { "05","08","02","06","09","01","07","33"};
```

```
    //bsort(bh.begin(), bh.end(),compdesc<string>); //普通函数，降序
    //bsort(bh.begin(), bh.end(), compasc<string>); // 普通函数，升序

    //bsort(bh.begin(), bh.end(), _creat<string>());    //仿函数，升序
    //bsort(bh.begin(), bh.end(), _less<string>()); //仿函数，降序，_less<string>()表示创建你匿名对象
    //bsort(bh.begin(), bh.end(), _less<int>());


    //bsort(bh.begin(), bh.end(), less<int>()); // STL提供的仿函数，升序
    //bsort(bh.begin(), bh.end(), greater<int>());  // STL提供的仿函数，降序

    //sort(bh.begin(), bh.end(), _creat<int>());    // STL提供的算法函数，升序，回调函数是自己写的仿函数
    //sort(bh.begin(), bh.end(), _less<int>()); // STL提供的算法函数，降序，回调函数是自己写的仿函数

    //sort(bh.begin(), bh.end(), less<int>());  // STL提供的算法函数，升序，回调函数是STL提供的仿函数
    sort(bh.begin(), bh.end(), greater<int>()); // STL提供的算法函数，降序，回调函数是STL提供的仿函数

    for (auto& val : bh)
    {
        cout << val << " ";
    }
    cout << endl;
}
```

STL提供的仿函数：

>	greater
<	less

STL提供的算法函数：sort()排序

函数声明：

template

void sort(RandomAccessIterator first, RandomAccessIterator last);

> sort()函数将[first，last]区间按升序排序，排序时使用值类型的<运算符进行比较。

函数声明：

template<class RandomAcceddIterator , class Compare>

void sort(RandomAccessIterator first, RandomAccessIterator last , Compare comp);

> sort()函数将[first，last]区间排序，排序时使用比较对象comp进行比较。

注意：

STL的sort算法，**数据量大时采用 QuickSort（快速排序）**，分段归并排序。一旦分段后的数据量小于某个门槛（16），为避免 QuickSort 的递归调用带来过大的额外负荷，就改用 InsertSort（插入排序）。**如果递归层次过深，就改用 HeapSort (堆排序)。**

适用于数组容器 vector，string，deque (list 容器有 sort成员函数，红黑树和哈希表没有排序的说法)