

第九天课堂笔记

一、回顾知识点

1.1 C++异常

抛出异常：

```
throw 异常信息
```

异常信息的数据类型： 基本数据类型、类、结构体等

限制抛出异常或声明异常接口：

在定义或声明函数时，在 {} 的左边，在() 右边声明函数内可能抛出的异常

```
返回值类型 函数名(参数列表) throw(异常类型, ...) {}
```

如果一个程序中出现了异常且未捕获处理，则会中断程序执行。

处理异常： 尝试捕获异常

```
try{
    可能存在异常的语句块;
}catch(异常类型 &error){
    处理异常的语句;
}catch(异常类型2或父类 &error){
    处理异常的语句;
}catch(...){
    // 以上未捕获的异常处理
}
```

【注意】catch属于精准基本数据类型的匹配，也可以进行类的多态匹配。

c++提供了标准了异常类： 基类 exception (包含了一个虚函数 what() 返回值为 const char *类)

```
logic_error: 逻辑错误， 构造函数带有string或char * 的参数
              invalid_argument
              out_of_range
              length_error
              ...
runtime_error: 运行时错误，构造函数带有string或char * 的参数
               overflow_error
               underflow_error
               range_error
bad_alloc:
...
```

自定义exception的派生类：

- 1) 继承exception类，可提供有参，也可以无参的构造函数， 需要重写父类的 what()函数。
- 2) 继承exception的子类

1.2 STL概念与三大组件的应用

STL (Standard Template Library, 标准模板库), C++ 引入了STL (C++自带的), 将数据结构和算法分离, 收录了很多高效的算法, 达到算法和特定的数据结构完美的结合。

STL六大组件:

容器: 各种的数据结构, `vector`, `list`, `set`, `map`, `queue`等
算法: 操作容器中的元素, 查询、存储、删除等相关的算法
迭代器: 算法 通过 迭代器 操作容器, 它是容器和算法的胶合剂
仿函数: 增强算法的功能, 依函数方式强化, `()`运算符重载
适配器: 扩展容器、算法、迭代器等接口
空间管理器: 空间的配置和管理

STL的三大组的应用

容器: 链表(`list`)、`vector`、`queue`(队列)、`stack` 栈, `set`集合, `map`映射表等
算法: 质变算法(改容器的元素内容)、非质变算法(查找容器的元素内容)
迭代器: 循环遍历容器中的每一个元素
输入迭代器、输出迭代器、向前迭代器、双向迭代器、随机迭代器

二、STL常见容器的应用

2.1 string容器

`string`也是字符串的类型, 可以理解时它是用于存放字符的容器。

2.1.1 String 和 c 风格字符串对比

- 1) `char *` 是一个指针, `string` 是一个类
- 2) `string` 封装了很多实用的成员方法
查找 `find`, 拷贝 `copy`, 删除 `delete` 替换 `replace`, 插入 `insert`
- 3) 不用考虑内存释放和越界
`string` 管理 `char*`所分配的内存。每一次 `string` 的复制, 取值都由 `string` 类负责维护, 不用担心复制越界和取值越界等。

2.1.2 string 容器api

```
// 构造函数
string(); //创建一个空的字符串 例如: string str;
string(const string& str); //使用一个 string 对象初始化另一个 string 对象
string(const char* s); //使用字符串 s 初始化
string(int n, char c); //使用 n 个字符 c 初始化

// 赋值
string& operator=(const char* s); //char*类型字符串 赋值给当前的字符串
string& operator=(const string &s); //把字符串 s 赋给当前的字符串
string& operator=(char c); //字符赋值给当前的字符串
string& assign(const char *s); //把字符串 s 赋给当前的字符串
string& assign(const char *s, int n); //把字符串 s 的前 n 个字符赋给当前的字符串
```

```

string& assign(const string &s); //把字符串 s 赋给当前字符串
string& assign(int n, char c); //用 n 个字符 c 赋给当前字符串
string& assign(const string &s, int start, int n); //将 s 从 start 开始 n 个
字符赋值给字符串

// 取值
char& operator[](int n); //通过[]方式取字符
char& at(int n); //通过 at 方法获取字符

// 字符串拼接
string& operator+=(const string& str); //重载+=操作符
string& operator+=(const char* str); //重载+=操作符
string& operator+=(const char c); //重载+=操作符
string& append(const char *s); //把字符串 s 连接到当前字符串结尾
string& append(const char *s, int n); //把字符串 s 的前 n 个字符连接到当前字符串结尾
string& append(const string &s); //同 operator+=( )
string& append(const string &s, int pos, int n); //把字符串 s 中从 pos 的 n 个字符连接到当前字符串结尾
string& append(int n, char c); //在当前字符串结尾添加 n 个字符 c

// 查找与替换
int find(const string& str, int pos = 0) const; //查找 str 第一次出现位置, 从 pos 开始查找, 如果未查找到返回 -1
int find(const char* s, int pos = 0) const; //查找 s 第一次出现位置, 从 pos 开始查找
int find(const char* s, int pos, int n) const; //从 pos 位置查找 s 的前 n 个字符第一次位置
int find(const char c, int pos = 0) const; //查找字符 c 第一次出现位置
int rfind(const string& str, int pos = npos) const; //查找 str 最后一次位置, 从 pos 开始查找
int rfind(const char* s, int pos = npos) const; //查找 s 最后一次出现位置, 从 pos 开始查找
int rfind(const char* s, int pos, int n) const; //从 pos 查找 s 的前 n 个字符最后一次位置
int rfind(const char c, int pos = 0) const; //查找字符 c 最后一次出现位置
string& replace(int pos, int n, const string& str); //替换从 pos 开始 n 个字符为字符串 str
string& replace(int pos, int n, const char* s); //替换从 pos 开始的 n 个字符为字符串 s

// 比较, 返回值 0:相等, 1:大于s, -1: 小于s
int compare(const string &s) const; //与字符串 s 比较
int compare(const char *s) const; //与字符串 s

// 截取子字符串
string substr(int pos = 0, int n = npos) const; //返回由 pos 开始的 n 个字符组成的字符串

// 插入与删除
string& insert(int pos, const char* s); //插入字符串
string& insert(int pos, const string& str); //插入字符串
string& insert(int pos, int n, char c); //在指定位置插入 n 个字符 c
string& erase(int pos, int n = npos); //删除从pos 开始的 n 个

// 转成 char *
const char * c_str() 将当前字符串转成 char *

// 获取字符串的长度

```

```
int size();
```

2.1.3 应用示例

如1： 创建两个字符串对象，将两个字符进行拼接第一个字符串，将拼接之后的字符串的'c'内容替换成'd'，查找替换之后字符串的内容 'is'第一次出现的位置和最后一次出现的位置。

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    string s1("disen is good!");
    string s2("xiaoma love color is black color!");

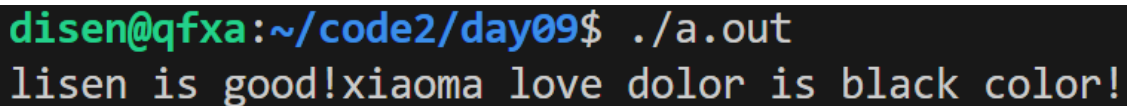
    // s1 += s2; // 拼接
    s1.append(s2);

    s1[0] = 'l'; // 修改指定位置的内容

    // 替换 第一个c为d
    int c_pos = s1.find('c');
    s1.replace(c_pos, 1, "d");

    cout << s1 << endl;

    return 0;
}
```



```
disen@qfxa:~/code2/day09$ ./a.out
lisen is good!xiaoma love dolor is black color!
```

【扩展】将拼接的字符串中的所有c替换成d

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    string s1("disen is good!");
    string s2("xiaoma love color is black color!");

    // s1 += s2; // 拼接
    s1.append(s2);

    // 替换所有的c为d
    int pos = s1.find('c');
    while (pos != -1)
    {
        s1.replace(pos, 1, "d");
        pos = s1.find('c', pos + 1); // 从上一次查找到的位置之后，再次查找
    }
}
```

```

    cout << s1 << endl;

    return 0;
}

```

```

disen@qfxx:~/code2/day09$ ./a.out
lisen is good!xiaoma love dolor is black dolor!
disen@qfxx:~/code2/day09$

```

需求的完整代码：

```

#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    string s1("disen is good!");
    string s2("xiaoma love color is black color!");

    // s1 += s2; // 拼接
    s1.append(s2);

    s1[0] = 'l'; // 修改指定位置的内容

    // 替换 第一个c为d
    int c_pos = s1.find('c');
    s1.replace(c_pos, 1, "d");

    cout << s1 << endl;

    // 查找is第一次和最后一次出现的位置
    int is_start = s1.find("is");
    int is_end = s1.rfind("is");
    cout << "is first=" << is_start << ", last=" << is_end << endl;

    // [扩展]将is第一次出现的位置和最后一次出现的位置的中间部分内容删除或替换成空格。
    s1.erase(is_start + 2, is_end - is_start - 2);
    cout << "删除之后的内容: " << s1 << endl;

    cout << s1.size() << endl;
    return 0;
}

```

```

lisen is good!xiaoma love dolor is black color!
is first=1, last=32
删除之后的内容: lisis black color!
18

```

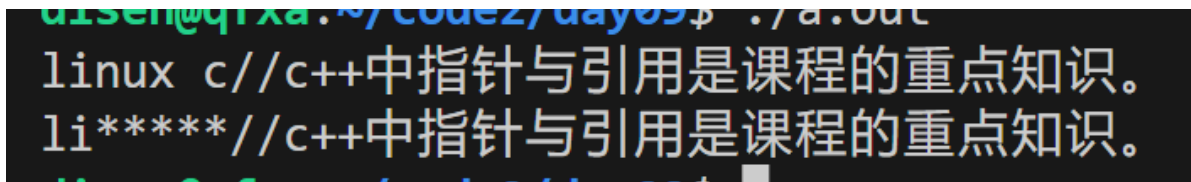
如2：将字符串 linux c/c++ 中指针与引用是课程的重点知识。 中 / 替换成 //，并删除第3的位置开始的5个字符。在第3的位置插入5个*。

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    string s1 = "linux c/c++中指针与引用是课程的重点知识。";
    s1.replace(s1.find('/'), 1, "//");
    cout << s1 << endl;

    s1.erase(2, 5);
    s1.insert(2, 5, '*');
    cout << s1 << endl;
    return 0;
}
```



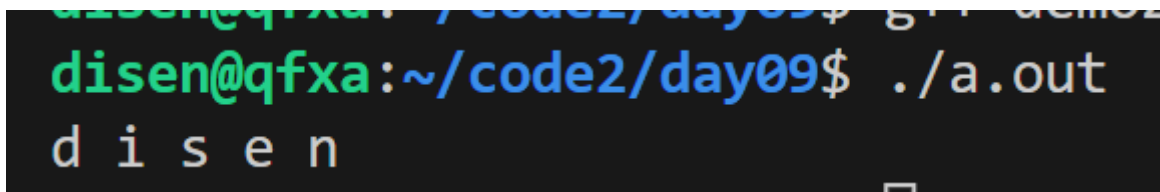
```
disen@qfxa:~/code2/day09$ ./a.out
linux c//c++中指针与引用是课程的重点知识。
li*****//c++中指针与引用是课程的重点知识。
```

如3: string使用迭代器遍历每一个字符

```
#include <iostream>

using namespace std;

int main(int argc, char const *argv[])
{
    string s2 = "disen";
    string::iterator it = s2.begin();
    while (it != s2.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
    return 0;
}
```

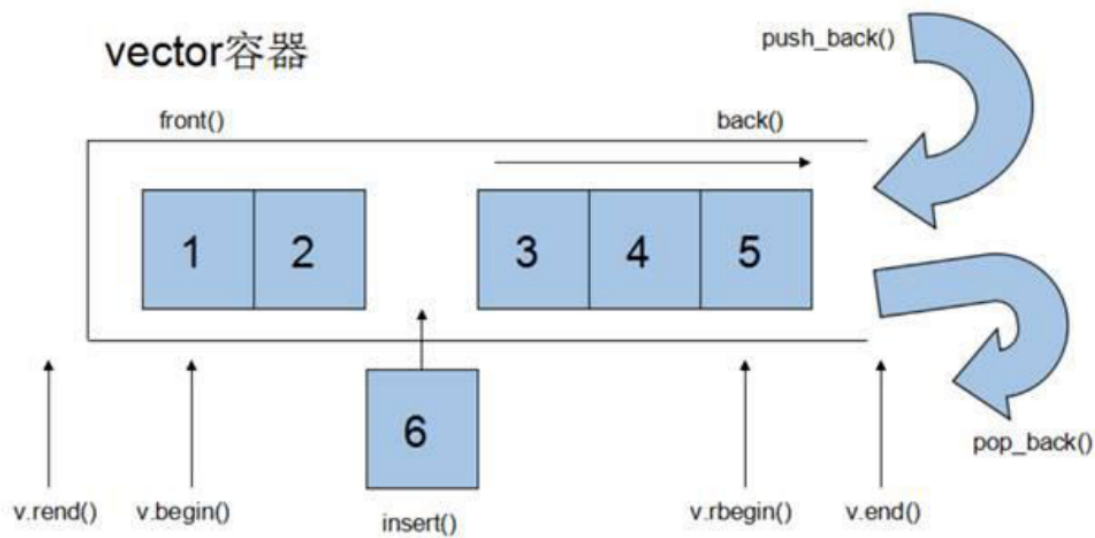


```
disen@qfxa:~/code2/day09$ ./a.out
d i s e n
```

2.2 vector容器

2.2.1 vector与数组的区别

vector的结构类同于数组，数组是静态的，在定义时确定的数组的大小；而vector是动态的，添加元素时如果空间不足时，则会自动扩容（ 2^n ）；整体来说，vector比较灵活的，而且vector是类模板，可以存放任意类型的元素。



2.2.2 vector 迭代器

vector 维护一个线性空间(线性连续空间), vector 迭代器所需要的操作行为是(*,->, ++, --, +, -, +=, -=)运算符重载等。vector 支持随机存取, vector 提供的是随机访问迭代器(Random Access Iterator), 迭代器中的元素即为vector容器中的元素。

如: 测试vector每一次扩容的大小

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

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v1;
    int mSize = v1.capacity();
    int cnt = 0;
    for (int i = 0; i < 1000; i++)
    {
        v1.push_back(i);
        if (mSize != v1.capacity())
        {
            mSize = v1.capacity();
            cout << "第" << ++cnt << "次, 扩容为 " << mSize << endl;
        }
    }
    return 0;
}
```

```
disen@qfxxa:~/code2/day09$ ./a.out
```

```
第1次, 扩容为 1
第2次, 扩容为 2
第3次, 扩容为 4
第4次, 扩容为 8
第5次, 扩容为 16
第6次, 扩容为 32
第7次, 扩容为 64
第8次, 扩容为 128
第9次, 扩容为 256
第10次, 扩容为 512
第11次, 扩容为 1024
```

【重要说明】: `vector` 容器扩容之后, 原迭代器则无效, 需要重新获取获取器

所谓动态增加大小, 并不是在原空间之后续接新空间(因为无法保证原空间之后尚有可配置的空间), 而是一块更大的内存空间, 然后将原数据拷贝新空间, 并释放原空间。

因此, 对 `vector` 的任何操作, 一旦引起空间的重新配置, 指向原`vector`的所有迭代器就都失效了。这是程序员容易犯的一个错误, 务必小心。

2.2.3 vector容器api

2.3.3.1 构造函数

```
// 构造函数
vector<T> v;    //采用模板实现类实现, 默认构造函数
vector(v.begin(), v.end()); //将 v[begin(), end())区间中的元素拷贝给本身。
vector(n, elem); //构造函数将 n 个 elem 拷贝给本身。
vector(const vector &vec); //拷贝构造函
```

如1: 指定元素及个数初始化vector

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

using namespace std;

int main(int argc, char const *argv[])
{
    vector<int> v1(5, 1);
    vector<int>::iterator it = v1.begin();
    while (it != v1.end())
    {
        cout << *it << endl;
        it++;
    }

    return 0;
}
```




如2： 使用数组的初始化vector

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

using namespace std;

int main(int argc, char const *argv[])
{
    int arr[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr, arr + 5);
    vector<int>::iterator it = v1.begin(); // 第一个元素的地址
    while (it != v1.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;

    return 0;
}
```

2.3.3.2 赋值操作

```
assign(beg, end); // 将 [beg, end) 区间中的数据拷贝赋值给本身。
assign(n, elem); // 将 n 个 elem 拷贝赋值给本身。
vector& operator=(const vector &vec); // 重载等号操作符
swap(vec); // 将 vec 与本身的元素互换
```

如： 使用数组的数据赋值给vector

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

using namespace std;

int main(int argc, char const *argv[])
{
    int arr[5] = {1, 2, 3, 4, 5};
    vector<int> v1;
    v1.assign(arr, arr + 5);
    vector<int>::iterator it = v1.begin(); // 第一个元素的地址
    while (it != v1.end())
    {
        cout << *it << " ";
    }
}
```

```

        it++;
    }
    cout << endl;

    return 0;
}

```

```

disen@qfxa:~/code2/day09$ g++ demo4.c
disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5

```

如：swap交换数据

```

#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    vector<int> v2(6, 1);
    print(v1);
    print(v2);
    v1.swap(v2);
    print(v1);
    print(v2);
    return 0;
}

```

```

disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5
1 1 1 1 1 1
1 1 1 1 1 1
1 2 3 4 5

```

2.3.3.3 大小操作

```
int size();    // 返回容器中元素的个数
bool empty();  // 判断容器是否为空， 返回bool值（0， 1）
void resize(int num); //重新指定容器的长度为 num，若容器变长，则以默认值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
void resize(int num, elem); //重新指定容器的长度为 num，若容器变长，则以 elem 值填充新位置。如果容器变短，则末尾超出容器长度的元素被删除。
int capacity(); //容器的容量
void reserve(int len); //容器预留 len 个元素长度，预留位置不初始化，元素不可访问。
```

如：resize() 调整容器的元素大小

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

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    cout << "size: " << v1.size() << endl;
    cout << "capacity: " << v1.capacity() << endl;
    cout << "first addr: " << &v1[0] << endl;
    v1.resize(10); // 只会改变容器中元素的个数
    cout << "first addr: " << &v1[0] << endl;
    cout << "size: " << v1.size() << endl;
    cout << "capacity: " << v1.capacity() << endl;
    return 0;
}
```

```
disen@qfxa:~/code2/day09$ ./a.out
size: 5
capacity: 5
first addr: 0x1e00c20
first addr: 0x1e01050
size: 10
capacity: 10
```

如：预留足够的空间，避免动态分配新的空间

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

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    v1.resize(10, 9); // 只会改变容器中元素的个数
    print(v1);

    vector<int> v2;
    v2.reserve(10); // 预留足够的空间，避免动态重新申请空间
    v2.push_back(1);
    cout << "first addr " << &v2[0] << endl;
    v2.push_back(2);
    cout << "first addr " << &v2[0] << endl;
    return 0;
}
```

```
first addr 0x20aac20
first addr 0x20ab080
disen@qfxa:~/code2/day09$ g++ demo5.cpp
disen@qfxa:~/code2/day09$ ./a
-bash: ./a: No such file or directory
disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5 9 9 9 9
first addr 0x825080
first addr 0x825080
disen@qfxa:~/code2/day09$
```

2.3.3.4 存取操作

```
at(int idx); //返回索引 idx 所指的数据，如果 idx 越界，抛出 out_of_range 异常。
operator[](int idx); //返回索引 idx 所指的数据，越界时，运行直接报错
front(); //返回容器中第一个数据元素
back(); //返回容器中最后一个数据元素
```

如：

```

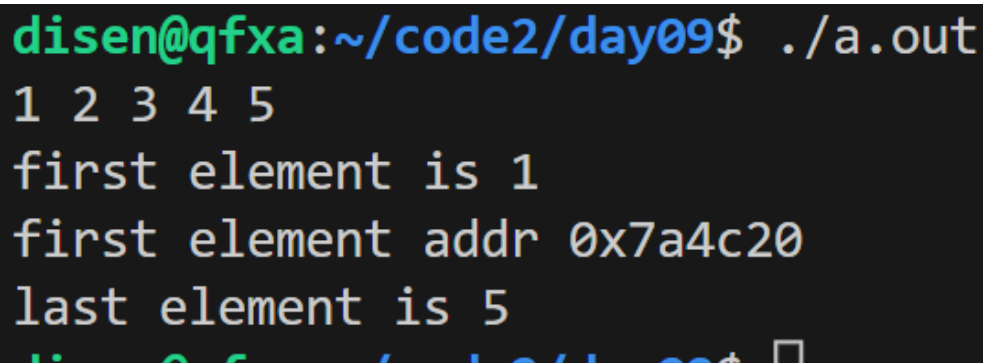
#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    print(v1);
    cout << "first element is " << v1.front() << endl;
    cout << "first element addr " << &(v1.front()) << endl;
    cout << "last element is " << v1.back() << endl;
    return 0;
}

```



```

disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5
first element is 1
first element addr 0x7a4c20
last element is 5

```

2.3.3.5 插入和删除

```

insert(const_iterator pos, int count, T ele); //迭代器指向位置 pos 插入 count个元素 ele.
push_back(ele); //尾部插入元素 ele
pop_back(); //删除最后一个元素
erase(const_iterator start, const_iterator end); //删除迭代器从 start 到 end 之间的元素, 删除[start, end)区间的所有元素
erase(const_iterator pos); //删除迭代器指向的元素
clear(); //删除容器中所有元素

```

如：插入元素

```

#include <iostream>
#include <vector>

using namespace std;

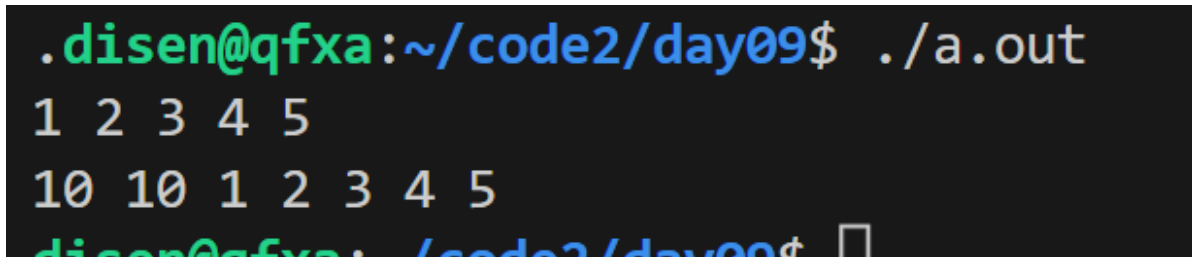
```

```

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    print(v1);
    vector<int>::iterator it = v1.begin();
    v1.insert(it, 2, 10); // 在当前it迭代的位置上插入2个10,之前的元素向后移动
    print(v1);
    return 0;
}

```



```

. disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5
10 10 1 2 3 4 5
disen@qfxa:~/code2/day09$

```

如：删除元素

```

#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{
    int arr1[5] = {1, 2, 3, 4, 5};
    vector<int> v1(arr1, arr1 + 5);
    print(v1);
    vector<int>::iterator it = v1.begin();
    v1.insert(it, 2, 10); // 在当前it迭代的位置上插入2个10,之前的元素向后移动
    print(v1);
    // 因为插入新元素，导致原空间释放了，则迭代器也无效了
}

```

```

    it = v1.begin();
    v1.erase(it);
    print(v1);
    vector<int>::iterator it_start = v1.begin() + 2;
    vector<int>::iterator it_end = v1.begin() + 4;

    v1.erase(it_start, it_end);
    print(v1);

    v1.clear();
    cout << "v1 empty() is " << v1.empty() << endl;
    print(v1);
    return 0;
}

```

```

disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5
10 10 1 2 3 4 5
10 1 2 3 4 5
10 1 4 5
v1 empty() is 1

```

2.2.4 小技巧示例

2.2.4.1 巧用 swap 收缩内存空间

resize()+swap()实现vector收缩内存空间

```

resize(n);
vector<T>(v).swap(v);

```

如:

```

#include <iostream>
#include <vector>

using namespace std;

void print(vector<int> &v)
{
    vector<int>::iterator it = v.begin();
    while (it != v.end())
    {
        cout << *it << " ";
        it++;
    }
    cout << endl;
}

int main(int argc, char const *argv[])
{

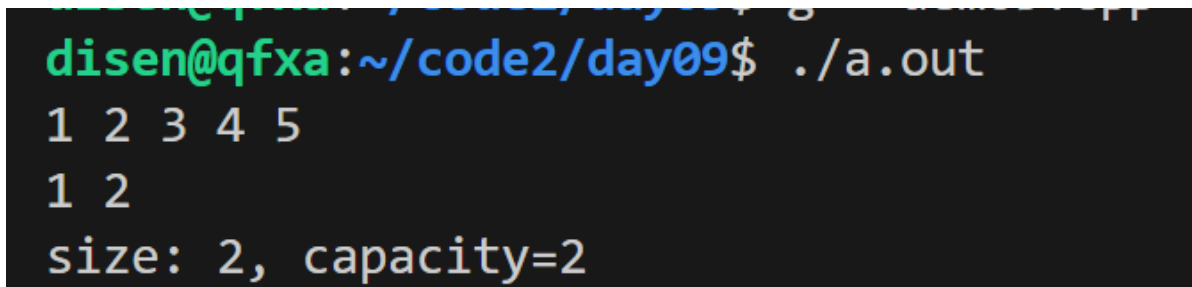
```

```

int arr1[5] = {1, 2, 3, 4, 5};
vector<int> v1(arr1, arr1 + 5);
print(v1);

v1.resize(2); // 缩小
vector<int>(v1).swap(v1);
print(v1);
cout << "size: " << v1.size() << ", capacity=" << v1.capacity() << endl;
return 0;
}

```



```

disen@qfxa:~/code2/day09$ ./a.out
1 2 3 4 5
1 2
size: 2, capacity=2

```

2.3 deque 容器

2.3.1 deque基本概念

2.3.2 deque实现原理

2.3.3 常用 API

2.3.3.1 构造函数

```

deque<T> deqT; //默认构造形式
deque(beg, end); //构造函数将[beg, end)区间中的元素拷贝给本身。
deque(n, elem); //构造函数将 n 个 elem 拷贝给本身。
deque(const deque &deq); //拷贝构造函数。

```

2.3.3.2 赋值操作

```

assign(beg, end); //将[beg, end)区间中的数据拷贝赋值给本身。
assign(n, elem); //将 n 个 elem 拷贝赋值给本身。
deque& operator=(const deque &deq); //重载等号操作符
swap(deq); // 将 deq 与本身的元素互换

```

2.3.3.3 大小操作

```

deque.size(); //返回容器中元素的个数
deque.empty(); //判断容器是否为空
deque.resize(num); //重新指定容器的长度为 num,若容器变长,则以默认值填充新位置。如果容器变短,则末尾超出容器长度的元素被删除。
deque.resize(num, elem); //重新指定容器的长度为 num,若容器变长,则以 elem 值填充新位置,如果容器变短,则末尾超出容器长度的元素被删除。

```


2.3.3.4 双端插入和删除

```
push_back(elem); //在容器尾部添加一个数据
push_front(elem); //在容器头部插入一个数据
pop_back(); //删除容器最后一个数据
pop_front(); //删除容器第一个数据
```

2.3.3.5 数据存取

```
at(idx); //返回索引 idx 所指的数据, 如果 idx 越界, 抛出 out_of_range。operator[]; //返回索引 idx 所指的数据, 如果 idx 越界, 不抛出异常, 直接出错。front(); //返回第一个数据。
back(); //返回最后一个
```

2.3.3.6 插入操作

```
insert(pos, elem); //在 pos 位置插入一个 elem 元素的拷贝, 返回新数据的位置。
insert(pos, n, elem); //在 pos 位置插入 n 个 elem 数据, 无返回值。
insert(pos, beg, end); //在 pos 位置插入 [beg, end) 区间的数据, 无返回值。
```

2.3.3.7 删除操作

```
clear(); //移除容器的所有数据
erase(beg, end); //删除 [beg, end) 区间的数据, 返回下一个数据的位置。
erase(pos); //删除 pos 位置的数据, 返回下一个数据的位置。
```

2.4 stack 容器

2.4.1 stack 基本概念

2.4.2 stack 没有迭代器

2.4.3 常用 API

2.4.3.1 构造函数

```
stack<T> stkT; //stack 采用模板类实现, stack 对象的默认构造形式:
stack(const stack &stk); //拷贝构造函数
```

2.4.3.2 赋值操作

```
stack& operator=(const stack &stk); //重载等号操作符
```

2.4.3.3 数据存取

```
push(elem); //向栈顶添加元素
pop(); //从栈顶移除第一个元素
top(); //返回栈顶元素
```

2.4.3.4 大小操作

```
empty();//判断堆栈是否为空  
size();//返回堆栈的大小
```