学习c++还必须掌握的

函数形参带默认值注意

1. 给默认值的时候，是从右向左给， --- 与 函数调用堆栈有关，从右往左压
2. 定义可以给出 默认值，声明也可以
3. 无论是定义给还是声明给，形参默认值只能是一次

4. 声明可以分段给

5. 声明分段给时，必须从右往左给

内联函数(inline)和普通函数区别

注意，只是建议，而不是强制

inline只有在release版本下有用，debug没用，还是需要 `call 函数` 指令

.o文件是看不到符号链接的

最大区别:普通函数调用有开销， 内联函数在编译阶段在函数调用点直接打开，直接省掉开销；内联函数成功将不会有函数符号；不是所有inline都会成为真正的内联函数

objdump -t .o 将不会看到符号链接

函数重载相关问题

1. 什么是函数重载?

函数重载是指同一作用域内，函数名相同但参数列表不同的函数

必须是在一个作用域，如果一个是main里的局部，一个是全局，不会发生重载

```
bool cmp(int a, int b){
    ...
}

bool cmp(const char* a, const char* b){
    ...
}

bool cmp(double a, double b){
    ...
}


main()    // 可以正确发生函数重载
{
    ...
    cmp(1,2);
    cmp(2.0,2.3);
```

```
}

main()    // 可以正确发生函数重载
{
    bool cmp(int a, int b);  // 加这个声明后，将不会正确产生重载，都将调用这个int的cmp
    ...
    cmp(1,2);
    cmp(2.0,2.3);
}
```

2. 面试常见坑

   一组函数，函数名相同,参数列表相同，仅返回值不同，不是!不是!重载!

3. const 或者 volatile 是怎么影响形参类型的?

   函数形参有无const，都是一个函数符号
   后续会讲，本节课没讲,先挖了个坑

4. 解释一下,什么是多态?

   静态多态---在编译时期的多态---常见:函数重载，模板
   动态多态---运行时期的多态---常见:**虚函数**和**继承**

5. cpp为什么支持函数重载，c不支持?

   C++在编译代码产生函数符号时，由函数名和参数列表类型组成，因此支持函数重载。

6. 函数重载的调用确定?

   在函数调用点，调用哪个函数重载版本在编译时期生成指令时就已确定。

7. cpp和c之间如何相互调用?

   cpp是无法直接调用c的代码的，因为符号链接 规则格式不同
   解决办法:

```
int sum(int a, int b);
// 改成
extern "C"
{
    int sum(int a, int b);  // 使用c写
}

// 高级写法:
#ifdef __cplusplus
```

```
extern "C" {
#endif

int add(int a, int b);

#ifdef __cplusplus
}
#endif
```

c同样无法直接调用cpp的代码的
给cpp文件的代码 加上 extern "C", 千万不能c文件里面, c没有那个东西

`typeid(变量名).name()` 是查看变量类型

const用法

1. const怎么理解?

   const修饰的变量 不能再作为左值!!初始化完成后, 不能被修改

2. c和cpp的const区别?

   C语言中, const修饰的变量称为常变量, 可以不初始化, 但最好初始化。
   不能作为常量使用

   C++中, const修饰的变量必须初始化, 否则编译不通过。
   C++编译时, const修饰的常量值会被替换, 因此可用于定义数组大小。
   若const变量的初始值是另一个变量, 则该const变量成为常变量, 与C语言中类似。

3. 为什么?

   编译方式不同!
   c中, const 是被当做一个常量 来编译生成指令的
   cpp中, 所有出现const常量名的地方, 都被常量的 初始化 替换了
   但是, 如果用变量 赋值给 const, 会退化为 常变量, 因为变量只有运行的时候, 才知道具体值, 因此 替换时,将不是替换常量值, 而是用 替换为变量
   也就无法 用于初始数组大小了

```
```

```c
int main() {
    const int a = 20;
    // a = 30; // 这行代码会导致编译错误
    int arr[a];  // c中会出错  cpp编译优化为int arr[10]
    int *p = (int*)&a;
    *p = 30;

    printf("%d\n%d\n%d\n", a, *p, *(&a));  // cpp编译优化为 20,*p, 20   cpp不会真正取地址解引用,编译阶段,视为常量, 直接替换
    return 0;
}
// c输出  30 30 30
// cpp输出 20 30 20

// 使用以下方式, 将和c无异, 30,30,30,
int b;
const int a=b;
```

4. 注意,c中const, 作为左值不能被修改, 但是 可以使用指针 修改内存上的内容

const与一二级指针结合

cpp中, 常量是绝对不能间接或直接 赋给 指针的!!

1. 在c中，由于const修饰的都叫做常变量，因此可以通过指针间接修改. 但是cpp中，const修饰作为常量使用时，不能通过一般指针间接修改值，为什么?

   因为使用一般指针时，是 int *和const int *的对应，而const int *在cpp中不能修改内存指向的值，只能指针重新指向新的内存。

   解决办法: 使用常量指针const int *作为指针

2. const与一级指针结合的两种情况?

   const修饰的是离他最近的类型!!!
   const int *p与 int const *p表示:const修饰的类型是int，修饰的表达式是*p，即p无法被更改，* p表示&a。

   注意是类型，*可不是类型!!

int* const p表示:const修饰的是类型是int，*修饰的表达式是p，即p是常量，即不能指向别的内存，* p可以被修改。

主要看const在*的前后去对比记忆!

以及双const: const int * const p

3. **cpp指针不使用NULL，而是nullptr**
4. 面试坑!

```
int a=10;
const int *p=&a;
int *q=p;
```

第三行类型转换是有错的！！

5. **一二级指针和const的类型转换公式？重点!**

`int * 《 const int *  右值赋给左值 这是不行的---错误的`

`const int * 《 int *可以，因为右边无限制---正确的`

`int *const 《 int*  左值实际是 int*, const右边无指针，不参与类型，实际是int * 《 int *---正确的`

`int ** 《int * const *，这实际还是与一级指针结合，两边都把int*去掉---- 错误的`

`int** 《 const int **  ----错误的，本质是与二级指针结合`

`const int** 《 int** ---错误的思考为什么，本质是与二级指针结合`

```
    int a=10;
    int *p=&a;
    const int * q=&p;

这是错误的因为，q和p指向的都是p所在的内存
而*p是const int *类型的（二级指针，两个*意义不同，第二个*表示的是这是指针量，第一个*结合前面的是表示类型），也就是说:

const int b=20;
*q = &b;
这在表面上看是可以的，因为左右类型相同，但是实际上，*q也是p，而p是int *普通指针，那些就变成了，int *  《  const int *，显然这是错误的！！
解决办法:那就一开始的p定义为const int *p，第二种方法，结合呗，const int *const*q呗，完美
```

int*const* 《 int** ---正确的 本质还是与一级指针结合

对于实际的const与二级指针结合，两边都必须有const！！！！

3．练习题注意看!

```
int*const*  《  const int**这么看:
   首先看const*《*可以
   然后int*《const int *  不可以
```

引用相关

1．**引用与指针的区别？重点**

引用是一种更安全的指针
引用必须初始化，指针不需要必须
从汇编看，指令是一模一样的
也正因此，引用初始化的右值必须能取地址，不能是数字啥的
引用无多级引用，只有一级引用

2. 引用数组 格式与 大小

sizeof(数组名)→整个数组大小， int*p=数组名，sizeof(p)→是指针大小，但是引用，却是整个数组大小
`int (&q)[5] = 数组名;`
sizeof(q) 是整个数组大小

切记!这是非法的--- `int &q = 数组名;`

3. 左值引用和右值引用?

这个概念，是被引用的值,是左值还是右值~
左值引用: `int a=10; int &b = a;` // a是左值，是有内存有名字的，值可以修改的
`int &b = 10;` // 10是右值，没内存，没名字--这是非法的

右值引用: c++11 提供了 : `int &&b = 10;` 从汇编看，是多了一步，把右值先存到一个栈空间(自动产生临时量)，然后 再给 引用
同时: `const int &b = 10;` 也是可以的，这两汇编是一样的， 但是区别是，const 修饰的 无法进行修改，正规右值引用，可以修改!!

一个右值引用变量，本身是个左值!!
右值引用变量,不能引用左值!!

const，一级指针,引用的结合使用

1. 写一个代码，在内存的0x00192827 写一个 4字节的 10?

```
int *p = (int *)0x00192827;//需要强转，本身是整数

//进化一下，内存值是整数,也是右值，则使用右值引用
int *&&p=(int *)0x00192827;
或者
int * const &p=(int *)0x00192827; // 不让地址变
```

2. **结合使用的 非合法 例子**

```
int a = 10;
int *p = &a;
int *&q = p;// 对不对?

int *&q = p; → int **q = &p;    是一模一样的, 用右边&覆盖左边第二个*
那 const int** ⇐ int **对吗, 显然不对


int a = 10;
int *const p = &a;
int *&q = p;// 对不对?

p是常量了, 那常量就不能赋给指针了!

```

new,delete, malloc, free区别

1. 区别?

   malloc和free 是 c的 库函数, 仅开辟内存, 不初始化.---malloc开辟内存失败,是通过返回值和nullptr作比较.

   而 new和delete 是 运算符! new可以在开辟内存是进行初始化.---new开辟失败,是抛出bad_alloc类型的异常来判断.

```
int *p = (int*) malloc(sizeof(int));
if(p == nullptr)
{
    return -1;
}
*p = 20;
free(p);


```

```
try
{
   int *p = new int(20);
}
catch(const std::bad_alloc &e)
{

}

数组:
int *p = (int*) malloc(sizeof(int)*20);
free(p);

int *p = new int[20]; // 中括号
int *p = new int[20](); // 合法的, 可以全部初始为0
int *p = new int[20](40); // 非法的, 不能给数组初始化具体值

delete[]p;  // 重点, 释放数组形式, 后面会讲,留意一下


```

2. new有几种?

```
new int(20);

//不抛出异常
new (nothrow) int;

const int* = new const int (20); // 常量

//定位new
new(&data) int (50); // 将指定内存 修改为整型, 并初始化
```

C++面向对象-类和对象那些你不知道的细节原理

学习c++还必须掌握的

- 函数形参带默认值注意
- 内联函数(inline)和普通函数区别
- 函数重载相关问题
- const用法
- const与一二级指针结合
- 引用相关
- const, 一级指针,引用的结合使用
- new,delete, malloc, free区别

C++面向对象-类和对象那些你不知道的细节原理

1. 类和对象, this指针
2. 构造函数和析构函数
3. 对象的深拷贝和浅拷贝
4. 类和对象应用实例--差一个循环队列
5. 掌握构造函数的初始化列表
6. 类的各种成员方法和区别
7. 指向类成员(成员变量和方法)的指针

c++模板编程-学习cpp类库的编程基础

1. 函数模板
2. 理解模板函数
3. 实现cpp的vector向量容器
4. 理解容器空间配置器allocator的重要性

运算符承载,是编程更灵活

1. 复数类comlex
2. string类
3. 迭代器iterator
4. vector容器 迭代器实现
5. 容器的迭代器失效问题
6. 深入理解new和delete原理
7. new和delete重载实现对象池应用

1.类和对象, this指针

1. OOP?

面向对象编程（OOP）是一种编程范式，核心思想是将数据和操作数据的方法封装在对象中，通过对象之间的交互来构建程序

2. 类?

   代表 实体的抽象类型(abstract data type, ADT).

3. 对象?

   实体→属性和行为，属性→成员变量，行为→成员方法.
   实例出来的类，就是对象

4. OOP四大特征?

   抽象，封装(隐藏)，继承，多态

5. 访问限定符?

   public, private, protected

6. 一般访问限定符用于什么?

   属性一般是私有的，
   共有的：一般用于给外部提供公有方法，访问私有属性

7. 类内的方法注意?

   类体内实现的方法，自动处理成 inline 内联函数

   类外定义方法，需要加 对应类的作用域： `void 类名::类方法名` ，类外方法将不是inline函数了，需要时,需手动添加inline.

8. vs注意:常量字符串问题?

   新的vs 不能使用 普通指针 `char *name` 接收"hzh"，需要使用常指针 `const char* name`

9. vs注意：一般不让用c类型的函数?

   在配置c++里，把 SDL检查关了

10. 在类外定义方法时，注意和c的不同?
    c中一般是整体作用域的方法，不加类空间，调用也不需要 加类名.方法名
    cpp却需要,为什么? 成员方法依赖于对象，比如你要看某个商品的 信息，哪个商品，就是哪个对象

11. c和cpp结构体区别，cpp的结构体和类区别?---注意分清楚

    c中结构体

    - 只能包含数据成员，不能包含函数（方法），
    - 通过外部函数来操作结构体，函数需要显式传递结构体指针或实例。
    - 没有访问控制（如 `public` 、 `private` ），所有成员默认是公开的

    cpp中 结构体和类

- 都可以包含数据成员和成员函数（方法）
- 结构体默认成员是 `public`，类默认成员是 `private`
- 可以直接在结构体或类中定义方法，并通过对象调用

12. 对象内存的大小?

    与成员方法无关

    只和 成员 变量有关，会自动对齐字节，取最大的成员变量作为基本，其他变量都开这么大
    vs里查看方法，在工具里打开vs的cmd，进入项目目录里，输入
    `dlreportSingleClassLayout类名`,即可看到

13. 字节对齐?

    先找最长的 ，然后进行整数倍对齐，不够则补位
    优点: 节省cpu访问内存的 io 次数

14. cpp的类注意
    每个对象有自己的成员，但是共享方法

15. 为什么需要this指针?一套成员方法，如何区分对象呢?
    编译器编译时，会给类的所有成员方法加一个this指针，这个不用手动输入，接收调用该方法的对象的地址

2.构造函数和析构函数

1. 自动初始化与结束自动执行，防止忘记类内函数.

    类名(){}------构造---可带参数，因此可以有多个构造函数，即重载
    ~类名(){}-------析构---不带参数!!，只能有一个----析构执行后，对象就不存在了，将不能再访问---不建议手动调用
    定义对象有两步: 开辟内存和调用构造

2. 注意多对象的构造与析构?
    先构造的后析构，后构造的先析构

3. 不同作用域的对象的 这两个函数如何调用?
    全局对象，定义的时候构造，程序执行结束，析构----因为在.data段
    堆上，无论什么时候，堆上的 析构，必须手动delete对象时，才会析构，而不会自动析构!!!

3.对象的深拷贝和浅拷贝

1. 拷贝构造?

```
class stack{};

stack s1;
stack s2=s1; //# 1
//stack s2(s1); //#2

#1和#2一样，都默认调用了拷贝构造函数，是做内存的浅拷贝， 都是初始化
delete不能释放野指针
当 对象占用外部资源，浅拷贝会出问题，浅拷贝会使得都指向 同一个外部资源(比如堆内存)，两次析构这块资源，肯定会出错

深拷贝的拷贝构造函数需要手写
stack(const stack &src) //传入对象
{
    外部资源需要重新指向新的，即开辟新的堆内存，并把数据复制过来
}


那对于
s2 = s1; // 这是赋值操作，这将也是 浅拷贝， 而且 还把s2 的原本外部资源丢了
本质:等号重载，s2.operator=(s1)
解决办法: 重写这个 重载函数
void operator(const stack &src) //传入对象
{
    if(thsi == &src)
    {
        return;  // 防止自己给自己赋值
    }
    先释放当前对象占用的外部资源
    然后根据src，重新开辟空间，即开辟新的堆内存，并把数据复制过来
}
```

2. 为什么不用memcpy函数，而是for循环放入？
   如果数组里只是简单数据，那是可以的，但是如果是对象，并且有指向外部资源的指针，那就有问题了，本质还是浅拷贝
3. 初始化拷贝，对象赋值，memcpy都是浅拷贝，需要注意！

4.类和对象应用实例--差一个循环队列

1. 实例-1

```
六、编写类String 的构造函数、析构函数和赋值函数（25 分）
已知类String 的原型为:
class String
{
public:
String(const char *str = NULL); // 普通构造函数
String(const String &other); // 拷贝构造函数
~ String(void); // 析构函数
String & operate =(const String &other); // 赋值函数
private:
char *m_data; // 用于保存字符串
};
请编写String 的上述4 个函数。
```

注意关闭sdl检查

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


class String
{
public:
    String(const char* str = nullptr) // 普通构造函数
  必须是const
    {
        if (str != nullptr)
        {
```

```
            m_data = new char[strlen(str) + 1];
            strcpy(this->m_data, str);
        }
        else
        {
            m_data = new char[1];
            *m_data = '\0';  // 防止传入nullptr时，后续都要判断
        }
    }
    String(const String& other)
    {
        m_data = new char[strlen(other.m_data) + 1];
        strcpy(m_data, other.m_data);
    }// 拷贝构造函数
    ~String(void)
    {
        delete[]m_data;
        m_data = nullptr;
    }// 析构函数

    String& operator=(const String & other)// 赋值函数 为什么返回String，而不是void，是为了支持多重赋值
    {
        if (this == &other)
        {
            return *this;    //单赋值，两个return去掉，返回void不影响
        }
        delete[]m_data;
        m_data = new char[strlen(other.m_data) + 1];
        strcpy(m_data, other.m_data);
        return *this;
    }
private:
    char* m_data; // 用于保存字符串
};

int main()
{
```

```
49      String s1;
50      String s2("hello");
51      String s3 = "hello";
52
53      String s4 = s3;
54      String s5;
55      s5 = s1 = s2;
56  }
```

5.掌握构造函数的初始化列表

1. 当一个类对象是另一个类的一部分，怎么去调用自定义的构造函数?

```
类1(....类2的对象需要的参数):类2的对象名(参数)
{
    ...
}

```

2. 类内成员初始化列表

```
类1(....类2的对象需要的参数):类2的对象名(参数), 类1的成员(值)...    还可以进行本类的成员的初始化
{
    ...
}

// 性能高,
//一般的是先定义出来, 再赋值,
//初始化列表是在定义的时候赋值
//普通类型在汇编可能没啥区别, 但是类对象的话, 区别很大
```

3. 构造函数的初始化列表
   可以指定当前对象成员变量的初始化方式，尤其是成员对象

4. 构造函数的初始化列表的 初始化顺序?

```
Test(int data=10):b(data), a(b){}

int a;
int b;

// 类似于上面的结构， a先定义， 则先初始化啊， 再初始化b
// a将是未分配值(win上会初始化为CC， -8589934460， linux则不会初始化为CC,不初始化)， b是10
```

6.类的各种成员方法和区别

内容： 常对象， 普通成员变量， 静态成员变量,静态方法， 普通方法， 常方法

1. 普通成员方法?

```
属于类的作用域
调用该方法时,需要依赖于对像
可以任意访问对象的私有成员变量
```

2. 静态成员变量， 类内声明， 类外定义并初始化
   必须,必须!
   静态成员变量 不算入对象的大小， 是所有对象共享的， 不属于对象， 而是属于类

```
类内私有:
static int count;  // 想要访问， 因为是私有的， 还需要写访问函数
...
类内变化:
count++;

类外:
int 类名::count = 0;
```

3. 静态成员方法，可以使用类名调用,而不是对象调用
   正如上面的count，访问的函数 可以写成静态成员方法，使用类名调用
   `类名::方法()`

```
属于类的作用域
用类名调用方法
可以任意访问对象的私有成员，但仅限不依赖于 对象的 成员(只能调用其他静态成员)
```

4. 普通成员方法和静态成员方法区别?
   普通的有this指针，这是本质，静态则没有this，不需要对象地址
   那这样就要注意，静态方法，不能访问普通成员变量，没有this指针

5. 常对象不能调用普通方法?---常成员方法
   究其本质，是普通方法的 this指针是 一般的 类*，但是常对象却是 const 类*，显然根据之前学的，这种转换是错误的
   怎么办?
   普通方法变为常成员方法，`原方法()const {}` 重新写个该方法，并加const即可，(复制完加const即可)，这也算是一种重载

6. 常成员方法---不修改成员变量的函数

```
属于类的作用域
调用依赖于对象，普通对象或常对象都行
```

7.指向类成员(成员变量和方法)的指针

```
类内:
int a;

类外:
int *p = &类::a; // 可以吗-----不可以，类::a 已经是 int 类::* 了，不是 int*了

解决办法:
int 类::*p = &类::a;  // 这是可以的
```

1. 注意，上面的这个p，将是类的，而不是对象的，因此他可以依赖于不同的对象，去使用

```
Test t1;
Test t2 = new Test();

int Test::*p = &Test::a;// 因为依赖于对象，所以必须有Test::*

t1.*p = 20;
t2→*p = 30;   //都是可以的，    依赖于对象

```

2. 如果 想指向静态成员变量呢?

```
    class Test
    {
        public:
            static int b;
    }
    int Test::b; // 千万不要忘记 静态的成员变量还需要再类外定义

    int *p1 = &Test::b;   // 这里注意，因为是类的一部分，在全局数据区，因此类型是int,而不是int Test::*
    *p = 30;   //由于不依赖于对象，所以可以直接用
```

3. 调用成员方法与之类似

```
普通方法: void(类名::*函数名)()=&类名::方法名;     (t1.*函数名)(); 这样调用
静态: void(*函数名)()=&类名::方法名;     (*函数名)(); 这样调用
```

c++模板编程-学习cpp类库的编程基础

学习c++还必须掌握的

- 函数形参带默认值注意
- 内联函数(inline)和普通函数区别
- 函数重载相关问题
- const用法
- const与一二级指针结合
- 引用相关
- const, 一级指针,引用的结合使用
- new,delete, malloc, free区别

C++面向对象-类和对象那些你不知道的细节原理

- 1.类和对象, this指针
- 2.构造函数和析构函数
- 3.对象的深拷贝和浅拷贝
- 4.类和对象应用实例--差一个循环队列
- 5.掌握构造函数的初始化列表
- 6.类的各种成员方法和区别
- 7.指向类成员(成员变量和方法)的指针

c++模板编程-学习cpp类库的编程基础

- 1.函数模板
- 2.理解模板函数
- 3.实现cpp的vector向量容器
- 4.理解容器空间配置器allocator的重要性

运算符承载,是编程更灵活

- 1.复数类comlex
- 2.string类
- 3.迭代器iterator
- 4.vector容器 迭代器实现
- 5.容器的迭代器失效问题
- 6.深入理解new和delete原理
- 7.new和delete重载实现对象池应用

1.函数模板

内容:

模板的实例化， 模板函数， 模板类型参数， 模板非类型参数， 模板的实参推演， 模板的特例化， 模板函数模板的特例化非模板函数的重载关系

> 区分　函数模板　和 模板函数的概念!!!!

1. 模板的意义?
   对类型也可以进行参数化了

```
// 原始的int的cmp函数
bool cmp(int a, int b)
{
    return a>b;
}
cmp(10, 20);

//现在的 模板函数
template<class或者typename T> //定义一个模板参数列表--尽量用template
bool cmp(T a, T b)  // 这是一个函数模板
{
    return a>b;
}

//使用
cmp<int>(10,20) --------- //函数调用点: 模板实例化为原始的 int 的 cmp函数
cmp<double>(10.4,20.5)
cmp(10,20) // 模板的实参推演
```

   在函数调用点, 编译器用用户指定的类型, 从原模板实例化一份函数代码出来(类型是变化的, int会出来int, double会出来double)

2. 模板的实参推演?
   根据用户传入的实参的类型, 推导出模板函数参数的具体类型

```
cmp(10.4,20)// 这是错误的

此时将不能使用模板的实参推演, 需要手动确定
cmp<int>(10.5,20)// double会转化为int
```

3. 函数模板是无法编译的, 因为不确定类型
   函数的实例化是在调用点进行
   模板函数才是编译器所编译的

4. 字符串是不能直接比较的
   const char * 用>比较，是比较内存值大小
   需要使用strcmp函数

5. 如果这个模板的类型是const char *，将需要进行模板特例化

```
// const char* 特例化
template<>  //定义一个模板参数列表--尽量用template
bool cmp<const char*>(const char* a, const char* b)
 // 这是一个函数模板
{
    return strcmp(a,b)>0;
}
```

6. 非模板函数--普通函数 以及调用关系

```
// 这是普通函数
bool cmp(const char* a, const char* b)  // 这是一个函数模板
{
    return strcmp(a,b)>0;
}

//调用关系
对比上个的特例化，二者存在时，cmp("aaa", "bbb")优先使用普通函数，编译器优先处理成普通函数符号，没有时，才会找特例化

```

7. 模板函数，模板的特例化，非模板函数的重载关系
   重载和模板一定要分清楚，有些书说，这是重载，重载是函数名相同，参数不同
   但要注意，模板的函数名，是函数名<类型>,这才是完整的函数名符号， 这个可不一样

8. 函数模板的声明和定义不能跨文件?

   当不在头文件时，而是普通的两个cpp文件：
   对于一般的函数模板，是不能把 声明和定义分开放置的，因为函数模板不参与编译，只有实例化后的模板函数 才会编译
   模板特例化是可以声明和定义分开放的，因为编译后有确定的 函数符号(UND)

定义和声明都放在头文件是可以的：
模板定义 放到头文件，声明放在主文件，因为include头文件 在预编译时，直接展开即可，所以可以看到模板定义的地方，即 定义和声明实际在一个文件

9. 那 8 的问题有办法吗？
有，直接声明时，指定类型-------尽量不要这么写

```
定义在头文件

声明这么写：
template bool cmp<int> (int, int); // double类似
```

2.理解模板函数

1. 模板的非类型参数
必须是 整数类型(整型或者地址,引用都可以)c++20之后好像可以浮点数了，只能使用,不能修改
指针和引用必须指向静态存储期的对象（如全局变量）

```
template<typeanem T, int size> // size是非类型参数
void sort(T *arr)
{
    排序代码...
}

//使用
int arr[]={....};
const int size = sizeof(arr)/sizeof(int);
sort<int, size>(arr);    // size在这里定义为是一个常量，可以使用具体数字代替


```

2. 类模板!---重点在于 类名到底是什么？
一定要注意：模板名称+类型参数列表=类名称
而不再是一般的 类名了，这会导致很多错误
类名<类型> 才是现在的 类名
类外定义方法，必须注意，这个作用域的问题

```
template<typename T>
class SeqStark    // 模板名称     +类型参数列表=类名称
{
public:
    //构造和析构函数名不需要加<T>, 其他出现的模板地方都加上类型
    SeqStark(int size=10);
    ~SeqStark();
    SeqStark<T>(const SeqStark<T> &stark); //拷贝构造函数
    SeqStark<T>& operator=(const const SeqStark<T> &stark);
.....

}
```

```
#include <iostream>
#include <stdexcept>

template<typename T>
class SeqStark {
private:
    T* data;         // 存储栈元素的数组
    int capacity;    // 栈的容量
    int top;         // 栈顶指针

public:
    // 构造函数
    SeqStark(int size = 10)
        : capacity(size)
        , top(-1)
        ,data = new T[capacity]{}

    // 析构函数
    ~SeqStark() {
        delete[] data;
    }

    // 拷贝构造函数
```

```
    SeqStark(const SeqStark<T> &other) : capacity(other.capacity), top(other.top) {
        data = new T[capacity]; // 类型不确定
        // 不要使用 memcpy, 如果是数组里是对象, 浅拷贝, 外部资源会出问题
        for (int i = 0; i ⩽ top; ++i) {
            data[i] = other.data[i];
        }
    }

    // 赋值运算符重载,  加了引用才能 支持链式赋值
    SeqStark<T>& operator=(const SeqStark<T> &other) {
        if (this ≠ &other) {
            delete[] data;
            capacity = other.capacity;
            top = other.top;
            data = new T[capacity];
            for (int i = 0; i ⩽ top; ++i) {
                data[i] = other.data[i];
            }
        }
        return *this;
    }

    // 压栈操作
    void push(const T& value) {
        if (top == capacity - 1) {
            throw std::overflow_error("Stack is full");
        }
        data[++top] = value;
    }

    // 弹栈操作
    void pop() {
        if (top == -1) {
            throw std::underflow_error("Stack is empty");
        }
```

```
        --top;
    }

    // 查看栈顶元素
    T& peek() const {
        if (top == -1) {
            throw std::underflow_error("Stack is empty");
        }
        return data[top];
    }

    // 判断栈是否为空
    bool isEmpty() const {
        return top == -1;
    }

    // 获取栈的大小
    int size() const {
        return top + 1;
    }
};

int main() {
    // 使用类模板创建一个整数栈
    SeqStark<int> intStack(5);

    // 压栈操作
    intStack.push(10);
    intStack.push(20);
    intStack.push(30);

    // 查看栈顶元素
    std::cout << "Top element: " << intStack.peek() << std::endl;

    // 弹栈操作
    intStack.pop();
    std::cout << "Top element after pop: " << intStack.peek() << std::endl;
```

```
96
97      // 判断栈是否为空
98      if (intStack.isEmpty()) {
99          std::cout << "Stack is empty" << std::endl;
100     } else {
101         std::cout << "Stack is not empty" <<
    std::endl;
102     }
103
104     // 获取栈的大小
105     std::cout << "Stack size: " << intStack.size()
    << std::endl;
106
107     return 0;
108 }
```

3. 类模板是　选择性的 实例化
   只有被调用的，才会实例化，具体看后续的代码调用，才会实例化
   类模板→实例化→模板类

3.实现cpp的vector向量容器

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

template<typename T>
class Vector
{
private:
    T* _first;//数组起始,与数组名
    T* _last;//数组最后位置的下一个
    T* _end;//空间的后面位置

    void expand() //二倍扩容
    {
        int size = _last - _first;
        T* ptmp = new T[2 * size];
        for (int i = 0; i < size; i++)
        {
            ptmp[i] = _first[i];
```

```
        }
        delete[]_first;
        _first = ptmp;
        _last = _first + size;
        _end = _first + 2 * size;
        // 这里注意堆内存和指向堆内存的指针
        //堆内存：一旦分配，会一直存在，直到显式释放
        /*
        如果指针是局部变量（比如在函数内部定义的），那么它的生命周期仅限于该函数的作用域。函数结束时，指针变量会被销毁，但它指向的内存不会被自动释放。

        如果指针是全局变量或类的成员变量，那么它的生命周期会与程序或对象的生命周期一致。
        */

    }

public:
    Vector(int size = 10)
    {
        _first = new T[size];
        _last = _first;
        _end = _first + size;
    }

    ~Vector()
    {
        delete[]_first;
        _first = _last = _end = nullptr;
    }

    Vector(const Vector<T>& src)
    {
        int size = src._end - src._first;
        _first = new T[size];
        int len = src._last - src._first;
        for (int i = 0; i < len; i++)
        {
            _first[i] = src._first[i];
```

```
        }
        _last = _first + len;
        _end = _first + size;
    }

    Vector<T>& operator=(const Vector<T>& src)
    {
        if (this == &src)
        {
            return *this;
        }

        delete[]_first;

        int size = src._end - src._first;
        _first = new T[size];
        int len = src._last - src._first;
        for (int i = 0; i < len; i++)
        {
            _first[i] = src._first[i];
        }
        _last = _first + len;
        _end = _first + size;
        return *this;

    }

    void push_back(const T& val) //向容器末尾添加元素
    {
        if (full())
        {
            expand();
        }
        *_last++ = val;
    }

    void pop_back() //向容器末尾删除元素
    {
        if (empty())
        {
```

```
            return;
        }
        -- _last;
    }

    bool full()const
    {
        return _last == _end;
    }

    bool empty()const
    {
        return _last == _first;
    }

    T back()const // 返回末尾元素
    {
        return *(_last-1);// *(--_last)错误的，本函数const方法，不能修改成员变量，_last-1是偏移量，--会改变last值
    }
};

int main()
{
    Vector<int> vec;
    for (int i = 0; i < 20; i++)
    {
        vec.push_back(rand() % 100);
    }

    while (!vec.empty())
    {
        cout << vec.back() << endl;
        vec.pop_back();
    }
    return 0;


}
```

```
135
```

4.理解容器空间配置器allocator的重要性

上一节的vector容器，当 类型是是下面这个：

```
class Test
{
public:
    Test()
    {
        cout<< "Test"<<endl;
    }
~Test()
    {
        cout<< "~Test"<<endl;
    }
}


//执行
Vector<Test> vect;  //会执行size次构造和析构，这是不合理的
```

1. 为什么会出现这个问题?
   因为new 会做两件事，开辟内存和构造Test对象，导致初始化Vector对象时，调用多次构造
   析构呢，应该是析构有限元素，而没有元素,析构是无意义的
2. 也引出了new和molloc的区别?

```
new:

适用于 C++ 中需要动态创建对象的场景。

支持构造函数和析构函数，适合面向对象编程。

malloc:

适用于 C 语言或需要直接操作内存的场景。

不涉及对象的构造和析构，适合底层内存管理。
```

3. 上文实现，在实际pop元素时，并没有析构这个对象.
4. 所以,现在需要做什么?
   首先要把内存开辟和Test对象构造分开处理----否则会造成空容器构造对象
   其次，要把析构对象和释放内存分开----从容器删除元素时，需要析构这个对象，因为可能占用外部资源
5. 容器空间配置器allocator?
   就做了3里面的事

```cpp
template<typename T>
class Allocator
{
    T* allocate(size_t size) 开辟内存
    {
        return (T*)malloc(sizeof(T)*size);
    }

    void deallocate(void *p) // 释放内存
    {
        free(p);
    }

    void  construct(T* p, const T &val) //对象构造
    {
        new (p) T(val); //定位new
        /*
```

```
        作用是在一块已经分配好的内存上构造一个对象，而不是通过 new 运算符动态分配内存。

p 是一个指针，指向一块预先分配好的内存。

T(val) 表示调用类型 T 的构造函数，并传递参数 val。

new (p) T(val) 的意思是在 p 指向的内存地址上构造一个 T 类型的对象，并调用构造函数 T(val)。

        */
    }

    void destroy(T *p) // 对象析构
    {
        p->~T(); //~T()代表T类型的析构函数
    }
}
```

6. 使用allocator---有点麻烦，慢慢看

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

template<typename T>
struct Allocator
{
    T* allocate(size_t size)// 开辟内存
    {
        return (T*)malloc(sizeof(T) * size);
    }

    void deallocate(void* p) // 释放内存
    {
        free(p);
    }

    void  construct(T* p, const T& val) //对象构造
    {
        new (p) T(val); //定位new
        /*
```

```
        作用是在一块已经分配好的内存上构造一个对象，而不是通过 new 运算符动态分配内存。

p 是一个指针，指向一块预先分配好的内存。

T(val) 表示调用类型 T 的构造函数，并传递参数 val。

new (p) T(val) 的意思是在 p 指向的内存地址上构造一个 T 类型的对象，并调用构造函数 T(val)。

        */
    }

    void destroy(T* p) // 对象析构
    {
        p->~T(); //~T()代表T类型的析构函数
    }
};

template<typename T, typename Alloc = Allocator<T>>
 //Alloc默认是Allocator<T>
class Vector
{
private:
    T* _first;//数组起始,与数组名
    T* _last;//数组最后位置的下一个
    T* _end;//空间的后面位置
    Alloc _allocator;//定义容器空间配置对象

    void expand() //二倍扩容
    {
        int size = _last - _first;
        //T* ptmp = new T[2 * size];
        T* ptmp = _allocator.allocate(2 * size);

        for (int i = 0; i < size; i++)
        {

            //ptmp[i] = _first[i];
```

```
            _allocator.construct(ptmp + i, _first[i]);
        }
        //delete[]_first;
        for (T* p = _first; p != _last; ++p)
        {
            _allocator.destroy(p); //析构_first指针指向的数组的有效元素
        }
        _first = ptmp;
        _last = _first + size;
        _end = _first + 2 * size;


    }

public:
    Vector(int size = 10)
    {
        //_first = new T[size];
        // 只开辟内存
        _first = _allocator.allocate(size);
        _last = _first;
        _end = _first + size;
    }

    ~Vector()
    {
        //delete[]_first;
        // 析构有效的元素并释放内存
        for (T* p = _first; p != _last; ++p)
        {
            _allocator.destroy(p); //析构_first指针指向的数组的有效元素
        }
        _allocator.deallocate(_first); //释放堆上的数组内存
        _first = _last = _end = nullptr;
    }

```

```
    Vector(const Vector<T>& src)
    {
        int size = src._end - src._first;
        //_first = new T[size];
        _first = _allocator.allocate(size);
        int len = src._last - src._first;
        for (int i = 0; i < len; i++)
        {
            //_first[i] = src._first[i];
            _allocator.construct(_first + i, src._first[i]);
        }
        _last = _first + len;
        _end = _first + size;
    }

    Vector<T>& operator=(const Vector<T>& src)
    {
        if (this == &src)
        {
            return *this;
        }

        //delete[]_first;

        for (T* p = _first; p != _last; ++p)
        {
            _allocator.destroy(p); //析构_first指针指向的数组的有效元素
        }

        int size = src._end - src._first;
        //_first = new T[size];
        _first = _allocator.allocate(size);
        int len = src._last - src._first;
        for (int i = 0; i < len; i++)
        {
            //_first[i] = src._first[i];
            _allocator.construct(_first + i, src._first[i]);
```

```
        }
        _last = _first + len;
        _end = _first + size;
        return *this;

    }

    void push_back(const T& val) //向容器末尾添加元素
    {
        if (full())
        {
            expand();
        }
        //*_last++ = val;
        _allocator.construct(_last, val);
        _last++;
    }

    void pop_back() //向容器末尾删除元素
    {
        if (empty())
        {
            return;
        }
        --_last;
        _allocator.destroy(_last);
    }

    bool full()const
    {
        return _last == _end;
    }

    bool empty()const
    {
        return _last == _first;
    }

    T back()const // 返回末尾元素
    {
```

```
170         return *(_last - 1);// *(--_last)错误的，本函数const方法，不能修改成员变量，_last-1是偏移量，--会改变last值
171     }
172 };
173
174 class Test
175 {
176 public:
177     Test()
178     {
179         cout << "Test" << endl;
180     }
181     ~Test()
182     {
183         cout << "~Test" << endl;
184     }
185     Test(const Test&)
186     {
187         cout << "Test(const)" << endl;
188     }
189 };
190
191 int main()
192 {
193     Test t1,t2;
194     cout << "----------" << endl;
195     Vector<Test> vec;
196
197     vec.push_back(t1);
198     vec.pop_back();
199     cout << "----------" << endl;
200     return 0;
201
202 }
203
```

运算符承载，是编程更灵活

1.复数类comlex

1. 定义复数类, 实现+的重载函数

```
1 # 复数类
```

```
*/
class CComplex
{
public:
    //这个构造函数可以有三种情形
    // (int, int) (int) ()
    CComplex(int r = 0, int i = 0)
    :mreal(r), minage(i) {}

    // 复数类与复数类相加
    CComplex operator+(const CComplex &src)
    {
        //CComplex comp;
        //comp.mreal= this->mreal+src.mreal;
        //comp.minage= this->minage+src.minage;
        //return comp;
        return CComplex(this->mreal+src.mreal,this->minage+src.minage );
    }

    //后置++
    CComplex operator++(int)
    {
        //CComplex comp;
        //comp.mreal= this->mreal+src.mreal;
        //comp.minage= this->minage+src.minage;
        //return comp;
        return CComplex(this->mreal++,this->minage++);
    }

    //前置++
    CComplex& operator++()
    {
        //CComplex comp= *this;
        mreal+=1;
        minage+=1;
        //return comp;
        return *this;
    }
```

```
    //+=
    CComplex& operator+=(const CComplex &src)
{
    this->mreal += src.mreal;
    this->minage += src.minage;
    return *this;
}

    void show(){...}
private:
    int mreal;
    int minage;
    friend CComplex operator+(const CComplex &lhs,
const CComplex &src);
     // 友元函数声明
    friend std::ostream& operator<<(std::ostream
&os, const CComplex &src);
};
//但是需要友元，类外访问私有，这个不是类成员方法，所以不需要
作用域
CComplex operator+(const CComplex &lhs, const
CComplex &src)
    {
        //CComplex comp;
        //comp.mreal= this->mreal+src.mreal;
        //comp.minage= this->minage+src.minage;
        //return comp;
        return
CComplex(lhs.mreal+src.mreal,lhs.minage+src.minage
);
    }

// 全局 operator<< 重载  注意，流不能用const修饰
std::ostream& operator<<(std::ostream &os, const
CComplex &src)
{
    os << "Real: " << src.mreal << ", Imaginary: "
<< src.minage;
    return os;
}
```

```
72
73
74  int main()
75  {
76      CComplex comp1(10, 10);
77      CComplex comp2(20, 20);
78      //需要+重载
79      CComplex comp3 = comp1 + comp2;
80
81      //与整型相加
82      CComplex comp4 = comp1 + 40;
83      /*
84      40是int，一般是要找对应的+重载：operator(int)→但是，编译器会先int转化为复数类，找有没有 CComplex(int) 的构造函数，而正好，构造函数的三种情形有这个→因此不用写重载了，编译器会使用构造函数把这个转化为复数类
85      */
86
87      // 编译器做对象运算时，优先调用对象的成员方法，如果没有，会在全局作用域找合适的
88
89
90      //下面这个整型加，需要在 全局作用域有重载函数
91      CComplex comp5 = 30 + comp2;//这是不行的，int在前，上一个可以 是因为编译器调用了 左边comp1的+，这个则没有
92
93      //operator++() 后置++        operator++(int) 后置++
94      comp5 = comp1++;
95      comp5 = ++comp1;
96
97      //+=重载
98      comp5 += comp1;
99
100     //cout<< 重载    ostream
101     // cin>>重载     istream
102     return 0;
103 }
```

2.string类

string更灵活简便，有+重载,> == < 重载，const char*可没有，需要调函数

仅供参考！

```cpp
#include <iostream>
#include <cstring>

class String
{
public:
    // 构造函数
    String(const char *p = nullptr)
    {
        if (p != nullptr)
        {
            _pstr = new char[strlen(p) + 1];
            strcpy(_pstr, p);
        }
        else
        {
            _pstr = new char[1];
            *_pstr = '\0';
        }
    }

    // 析构函数
    ~String()
    {
        delete[] _pstr;
        _pstr = nullptr;
    }

    // 拷贝构造函数
    String(const String &other)
    {
        _pstr = new char[strlen(other._pstr) + 1];
        strcpy(_pstr, other._pstr);
```

```
34      }
35
36      // 赋值运算符重载
37      String &operator=(const String &other)
38      {
39          if (this == &other) // 防止自赋值
40          {
41              delete[] _pstr;
42              _pstr = new char[strlen(other._pstr) + 1];
43              strcpy(_pstr, other._pstr);
44          }
45          return *this;
46      }
47
48      // 加法运算符重载--未优化，多了一次new,delete
49      String operator+(const String &other) const
50      {
51          char *newtmp = new char[strlen(_pstr) +
    strlen(other._pstr) + 1];
52          strcpy(newtmp, _pstr);
53          strcat(newtmp, other._pstr);
54          String newString(newtmp);
55          delete[]newtmp;
56          return newString;
57      }
58
59      // 加法运算符重载--小优化后
60      String operator+(const String &other) const
61      {
62          String newString;
63          newString._pstr = new char[strlen(_pstr) +
    strlen(other._pstr) + 1];
64          strcpy(newString._pstr, _pstr);
65          strcat(newString._pstr, other._pstr);
66          return newString;
67      }
68
69      // 比较运算符重载
70      bool operator>(const String &other) const
71      {
```

```
72          return strcmp(_pstr, other._pstr) > 0;
73      }
74
75      bool operator<(const String &other) const
76      {
77          return strcmp(_pstr, other._pstr) < 0;
78      }
79
80      bool operator==(const String &other) const
81      {
82          return strcmp(_pstr, other._pstr) == 0;
83      }
84
85      // 长度方法
86      size_t length() const
87      {
88          return strlen(_pstr);
89      }
90
91      // 下标运算符重载
92      char &operator[](size_t index)
93      {
94          return _pstr[index];
95      }
96
97      const char &operator[](size_t index) const
98      {
99          return _pstr[index];
100     }
101
102     // 输出运算符重载
103     friend std::ostream &operator<<(std::ostream &os,
    const String &str)
104     {
105         os << str._pstr;
106         return os;
107     }
108
109     // 输入运算符重载
```

```
    friend std::istream &operator>>(std::istream &is, String &str)
    {
        char buffer[1024];
        is >> buffer;
        str = String(buffer);
        return is;  // 支持链式操作  例如: cin>>a>>b,第一次cin>>a返回cin, 即后面的变为 cin>>b
    }

private:
    char *_pstr;
};

int main()
{
    String str1 = "Hello";
    String str2 = "World";
    String str3 = str1 + " " + str2;

    std::cout << str3 << std::endl; // 输出: Hello World

    if (str1 > str2)
        std::cout << "str1 is greater than str2" << std::endl;
    else if (str1 < str2)
        std::cout << "str1 is less than str2" << std::endl;
    else
        std::cout << "str1 is equal to str2" << std::endl;

    std::cout << "Length of str1: " << str1.length() << std::endl;

    return 0;
}
```

3.迭代器iterator

继续优化 operator+

1. 对于原始的stl的string类，使用迭代器，一个个 输出

```
String str1 = "hello hzh";
string::iterator it = str1.begin(); //或者auto
auto it = str1.begin();
for(; it≠str1.end();++it)
{
    cout << *it <<" ";
}
```

2. 迭代器可以透明的访问 容器内部的 元素的值，不需要考虑 类型
3. 泛型算法--全局的函数---给所有容器用的
4. 泛型算法，有一套方式，能够统一的遍历所有容器的元素--迭代器
5. 写一个自定义的 迭代器，嵌套在上一个自定义的String类

```
// 自定义迭代器
    class iterator
    {
    public:
        iterator(char *ptr = nullptr) : _ptr(ptr) {}

        // 解引用操作符
        char &operator*() const
        {
            return *_ptr;
        }

        // 前置递增操作符
        iterator &operator++()
        {
            ++_ptr;
            return *this;
        }

```

```
        // 后置递增操作符
        iterator operator++(int)
        {
            iterator tmp = *this;
            ++_ptr;
            return tmp;
        }

        // 相等操作符
        bool operator==(const iterator &other) const
        {
            return _ptr == other._ptr;
        }

        // 不相等操作符
        bool operator≠(const iterator &other) const
        {
            return _ptr ≠ other._ptr;
        }

    private:
        char *_ptr;
    };

    // 返回指向字符串开头的迭代器
    iterator begin()
    {
        return iterator(_pstr); //这样写不能引用哈，因为是局部变量
    }

    // 返回指向字符串结尾的迭代器
    iterator end()
    {
        return iterator(_pstr + length());
    }

private:
    char *_pstr;
};
```

```
59
```

6. c++11里,有方便的 迭代器调用方式:

```
for(char ch : str1)
{
    cout << ch <<" ";
}
```

7. 迭代器功能:
   提供一种统一的方式, 来透明遍历容器

4.vector容器 迭代器实现

```
    // 自定义迭代器
    class iterator
    {
    public:
        iterator(T* ptr = nullptr) : _ptr(ptr) {}

        // 解引用操作符
        T& operator*() const
        {
            return *_ptr;
        }

        // 前置递增操作符
        iterator& operator++()
        {
            ++_ptr;
            return *this;
        }

        // 后置递增操作符
        iterator operator++(int)
        {
            iterator tmp = *this;
            ++_ptr;
```

```
            return tmp;
        }

        // 相等操作符
        bool operator==(const iterator& other) const
        {
            return _ptr == other._ptr;
        }

        // 不相等操作符
        bool operator≠(const iterator& other) const
        {
            return _ptr ≠ other._ptr;
        }

    private:
        T* _ptr; // 指向当前元素的指针
    };

    // 返回指向容器开头的迭代器
    iterator begin()
    {
        return iterator(_first);
    }

    // 返回指向容器末尾的迭代器
    iterator end()
    {
        return iterator(_last);
    }
```

对于vector，是可以用下标的

5.容器的迭代器失效问题

新学两个 容器 的函数，添加和删除
容器对象.insert(it, val)　容器对象.erase(it)　--这两都是 要传入迭代器!!

1. 迭代器失效-1

```
//删除所有的偶数
for(; it≠vec.end(); ++it)
{
    if(*it %2 ==0)
    {
        // 第一次调用erase后，it就失效了，不能再++了，
        vec.erase(it);
    }

}
```

2. 迭代器失效-2

```
//在所有偶数前面添加一个小于偶数值1的值
for(; it≠vec.end(); ++it)
{
    if(*it %2 ==0)
    {
        // 第一次调用insert后，it就失效了
        vec.insert(it, *it-1);
    }

}
```

3. 迭代器为什么会失效?
    1. 删除(erase)或增加(insert)it的地方后，当前位置及后续的迭代器全部失效，但是之前的仍然有效
    2. insert如果引起容器扩容，会整体全部失效，不是一块内存了
    3. 不同容器迭代器不能进行比较

4. stl的容器，删除后解决办法，for里不要++，并更新 迭代器，当前位置it

```
//删除所有的偶数
for(; it≠vec.end();)
{
    if(*it %2 ==0)
    {
        // 第一次调用erase后，it就失效了，不能再++了，
        it = vec.erase(it);
    }
    else
    {
        ++it;
    }

}
```

5. stl的容器,增加但不扩容，解决办法，要+两次

```
//在所有偶数前面添加一个小于偶数值1的值
for(; it≠vec.end(); ++it)
{
    if(*it %2 ==0)
    {
        // 第一次调用insert后，it就失效了
        vec.insert(it, *it-1);
        ++it;
    }
}
```

6. 迭代器失效原理?
   vector 解决失效的 代码 -整体使用链表来存储迭代器,-- 这也就导致了 链表节点处对不上，将会失效
7. 了解 原理，知道什么时候会失效，其余的代码，讲的有点乱--网上看看

6.深入理解new和delete原理

1. new和delete 本质是 运算符重载
   从汇编看，会调用 operator new 和 operator delete
2. 回顾1.8节，new，delete，malloc，free 区别
   malloc按字节开辟内存;new开辟需要指定类型 new int[10]
   so，malloc 开辟内存返回的都是 void*，operator new→ int*
   malloc只负责开辟，new不仅开辟，还初始化
   malloc 错误返回 nullptr，new抛出bad_alloc类型异常

   delete：调用析构，再free，　free：仅释放内存
3. new实现--简化版

```cpp
//先开辟内存，再调用对象构造函数
void* operator new(size_t size)
{
    void *p = malloc(size);
    if(p == nullptr)
        throw bad_alloc();
    return p;
}
//调用 对象的 析构，再释放
void operator delete(void* p) noexcept
{
    if (p != nullptr) {
        free(p);  // 释放由 malloc 分配的内存
    }
}
//delete 操作符通常被标记为 noexcept，表示它不会抛出异常。这是为了确保在析构对象时不会因为内存释放失败而抛出异常。


// 自定义 new[] 操作符
void* operator new[](size_t size)
{
    void* p = std::malloc(size);  // 使用 malloc 分配内存
    if (p == nullptr) {
        throw std::bad_alloc();  // 如果分配失败，抛出 bad_alloc 异常
```

```
    }
    return p;
}

// 自定义 delete[] 操作符
void operator delete[](void* p) noexcept
{
    std::free(p);  // 使用 free 释放内存
}


int main()
{
    try
    {
        int *p=new int;
        delete p;
    }
    catch(const bad_alloc &err)
    {
        cerr << err.what() << endl;
    }
}
```

4. new和delete能混用吗？ cpp为什么要区分 单个元素释放和 数组释放?---面试重点
   new/delete
   new[]/delete[]
   对于普通的 编译器内置类型(int等)，可以混用，new/delete[]
   new[]/delete
   对于自定义类 类型，有析构，为了调用正确的析构，开辟对象数组的时候，会多开辟4个字节，记录对象的个数，混用会导致 无法正确释放 这多出来的4字节

5. 为什么自定义类，需要额外开辟?
   因为普通类型的大小是固定的，编译器可以直接计算。`delete` 不需要额外信息来释放内存，

6. 自己补充的剖析：那么为什么,普通类型不需要?
   本质是析构的问题， new/delete 不会主动计算有几个对象的
   对于一般类型，new[]开辟一个完整的内存块，由于没有析构，不需要遍历，所以直接释放即可

而有析构的类，在delete时，需要一个个遍历 析构函数，而编译器不知道数组里有几个对象，需要遍历几次，因此必须开辟一个额外的空间,存储有几个对象

7.new和delete重载实现对象池应用

对象池：对象复用
对象池是一种通过预先创建并重复使用对象来减少创建和销毁开销，从而提升性能的设计模式。

1. 对于这个程序，会大量调用 new和delete，影响性能

```
template<typename T>
class Queue
{
public:
    Queue()
    {
        _front = _rear = new QueueItem();
    }

    //析构需要遍历
    ~Queue()
    {
        while (_front != nullptr)
        {
            QueueItem* temp = _front;
            _front = _front->_next;
            delete temp;
        }
    }

    // 添加入队元素
    void push(const T& data)
    {
        QueueItem* newItem = new QueueItem(data);
        _rear->_next = newItem;
```

```
        _rear = newItem;
    }

    // 出队操作
    bool pop(T& data)
    {
        if (_front == _rear)
        {
            return false; // 队列为空
        }

        QueueItem* temp = _front->_next;
        data = temp->_data;
        _front->_next = temp->_next;

        if (_rear == temp)
        {
            _rear = _front; // 如果出队的是最后一个元素，重置
_rear
        }

        delete temp;
        return true;
    }

private:
    struct QueueItem
    {
        QueueItem(T data = T()) : _data(data),
_next(nullptr) {}
        T _data;
        QueueItem* _next;
    };

    QueueItem* _front; // 指向头节点
    QueueItem* _rear;  // 指向队尾
};
```

```cpp
int main()
{
    Queue<int> q;

    q.push(10); //大量调用 new和delete, 每次新元素都要new
    q.push(20);
    q.push(30);

    int data;
    while (q.pop(data))
    {
        std::cout << "Dequeued: " << data << std::endl;
    }

    return 0;
}
```

解决办法: 使用对象池, new和delete重载, 使得push不需要开辟新的

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

#define POOL_ITEM_SIZE 100000

template<typename T>
class Queue
{
private:
    struct QueueItem
    {
        T _data;
        QueueItem* _next;
        static QueueItem* _itemPool;

        QueueItem(T data = T()) : _data(data), _next(nullptr) {} //零构造

        // 重载 operator new, 使用对象池管理内存
        void* operator new(size_t size)
```

```
        {
            if (_itemPool == nullptr)
            {
                // 预分配 POOL_ITEM_SIZE 个 QueueItem
                _itemPool = reinterpret_cast<QueueItem*>(new char[POOL_ITEM_SIZE * sizeof(QueueItem)]);
                QueueItem* p = _itemPool;
                for (0; p < _itemPool + POOL_ITEM_SIZE - 1; ++p)
                {
                    p->_next = p + 1; // 链接对象池中的空闲对象
                }
                p->_next = nullptr; // 结束链表
            }

            // 从对象池取出一个对象
            QueueItem* p = _itemPool;
            _itemPool = _itemPool->_next;
            //p->_next = nullptr; // 防止误用
            return p;
        }

        // 重载 operator delete，将对象归还到池中
        void operator delete(void* ptr)
        {
            QueueItem* obj = static_cast<QueueItem*>(ptr);
            obj->_next = _itemPool;
            _itemPool = obj;
        }
    };

    QueueItem* _front; // 头指针（哨兵）
    QueueItem* _rear;  // 尾指针

public:
    Queue()
    {
```

```
55          _front = new QueueItem(); // 创建哨兵节点
56          _rear = _front;           // 初始时 rear 指向
    front
57          cout << "Queue" << endl;
58      }
59
60      ~Queue()
61      {
62          while (_front != nullptr)
63          {
64              QueueItem* temp = _front;
65              _front = _front->_next;
66              delete temp; // 归还到对象池
67          }
68          cout << "~Queue" << endl;
69      }
70
71      // 入队操作
72      void push(const T& data)
73      {
74          QueueItem* newItem = new QueueItem(data); // 从
    对象池获取, new重载了
75          _rear->_next = newItem;
76          _rear = newItem;
77      }
78
79      // 出队操作
80      bool pop(T& data)
81      {
82          if (_front->_next == nullptr)
83          {
84              return false; // 队列为空
85          }
86
87          QueueItem* temp = _front->_next;
88          data = temp->_data;
89          _front->_next = temp->_next;
90
91          if (_rear == temp)
92          {
```

```
            _rear = _front; // 如果出队的是最后一个元素，重置 _rear
        }

        delete temp; // 归还到对象池
        return true;
    }
};

// 初始化静态成员变量
template<typename T>
typename Queue<T>::QueueItem*
Queue<T>::QueueItem::_itemPool = nullptr;

int main()
{
    Queue<int> q;
    q.push(10);
    q.push(20);
    q.push(30);

    int value;
    while (q.pop(value))
    {
        std::cout << "Popped: " << value << std::endl;
    }

    return 0;
}
```