38.static的用法和作用
39.形参和实参的区别
40.值传递，指针传递，引用传递
41.静态变量的初始化？
42.const关键字的作用
43.类的继承
44.汇编层解释引用
45.delete， delete[] allocator的作用
46.new和delete原理，delete怎么知道释放内存大小
47.malloc申请的存储空间能用delete释放么
48.malloc和free实现原理（太底层）
49.malloc realloc calloc的区别
50.类成员初始化？构造函数执行程序？成员初始化列表更快？
51.哪些情况必须列表初始化？作用是什么？
52.char*和string的区别，如何实现
53.什么是内存泄漏？检测与避免？
54.对象复用的了解，零拷贝的了解
55.介绍面向对象的三大特性，举例说明
56.四种强制类型转换
57.C++函数调用的压栈过程
58.C++中的一类错误 coredump，怎么调试？
59.移动构造函数？
60.怎么获得结构成员相对于结构开头的字节偏移量
61.静态类型和动态类型？动态绑定和静态绑定？
62.引用能否实现动态绑定？
63.全局变量和局部变量
64.指针加减计算要注意什么？
65.怎么判断两个浮点数相等？
66.方法调用的原理
67.C++的传指针和传引用的区别和原理？
68.类如何实现只能静态分配和只能动态分配？
69.如果将某个类用作基类，为什么必须定义而非声明？
70.继承中对象如何转换？指针和引用如何转换？
71.C++的组合？它和继承相比有什么优缺点？
72.函数指针
73.说一下你理解的内存对齐以及原因
74.结构体变量比较是否相等
75.函数调用过程栈的变化，返回值和参数变量哪个先入栈？
76.define,const,typedef,inline的使用方法和区别？
77.printf的实现原理

78、为什么模板类一般都是放在一个h文件中
79.C++类成员的访问权限和继承权限问题
80.cout和printf
81.重载运算符？函数匹配的顺序和原则？
82.定义和声明的区别
83.全局变量和static变量的区别
84.静态成员与普通成员的区别
85.说一下你理解的ifdef endif代表着什么？
86.隐式类型转换，如何消除
87.C++如何处理多个异常？
88.不使用额外空间交换两个数？
89.strcpy和memcpy
90.程序执行int main(int argc, char *argv[])时的内存结构
91.volatile关键字的作用
92.如果有一个空类，会默认添加哪些函数？
93.C++中标准库是什么
94.const char* 和 string的关系
95.什么时候用指针当参数，什么时候用引用？
96.设计一个计算仅单个子类的对象个数
97.成员初始化列表在什么时候用到？调用过程是什么？
98.传引用的好处？
99.说一说strcpy、sprintf与memcpy这三个函数的不同之处
100. 引用作为函数的好处？
101.数组和指针的区别？
102.阻止一个类被实例化的方法？
103.如何禁止程序自动生成拷贝构造？
104.Debug和Release的区别
105.main函数的返回值
106.strcpy和strncpy的区别
107.static_cast比C语言中的转换强在哪里
108.成员函数里memset(this,0,sizeof(*this))
109. 回调函数？作用？
110.什么是一致性哈希？
111.C++从代码到可执行程序经历了什么？
    1.预编译，处理#开头的语句 4个#
    2.编译
    3.汇编
    4.链接
112.友元函数和友元类
113.动态编译和静态编译

114.hello.c的编译过程

115.介绍一下几种典型的锁（还得多看）

读写锁

互斥锁

条件变量

自旋锁

116.delete和delete[]

117.为什么不能把所有的函数写成内联函数

118.C++为什么没有垃圾回收机制？和JAVA不一样

二、内存管理和新标准

1.类的对象存储空间

2.简要说明C++的内存分区

3.什么是内存池，如何实现

4.你了解的C++内存管理？

5.C++中类的数据成员和成员函数内存分布情况

6.关于this指针知道什么？

7.几个this的易混问题

1.创建时间

2.this指针放在哪里？

3.this指针如何传递类中的指针？绑定么，还是函数参数的首参数就是this指针？this指针如何找到“类实例化后的函数的”？

4.this指针是如何访问类中的变量？

5.我们能直接使用一个对象this指针？

6.每个类编译后，是否创建一个类中函数表保存函数指针，以便调用函数？

8.内存泄漏的后果？监测？解决方法？

9.在成员函数中调用delete this会出现什么问题？对象还能使用么？

10.为什么是不可预期的问题？

11.在类的析构中调用delete this，会发生什么？

12.空类的大小是多少？

13.说下几种情况下的类的大小是多少？

14.this指针调用成员变量时，堆栈会发生什么变化？

15.类对象的大小受哪些因素影响？

C++11新标准

16.新特性有哪些

17.auto decltype 和decltype(auto)

18.NULL和nullptr的区别

19.智能指针的原理，常见的智能指针及实现

20.lambda函数

21.智能指针的作用

22.了解的auto_ptr

23.智能指针的循环引用

24.手写实现智能指针类需要实现哪些函数？

25.循环引用怎么办？

三、STL模板库和其他问题

STL模板库

1.STL？

2.trivial destructor

3.使用智能指针管理内存资源，RAII是怎么回事？

4.迭代器：++it和it++哪个好？

5.说一下C++左值引用和右值引用

6.STL中hashtable的实现

7.简单说一下traits技法

8.STL的两级空间配置器

一级空间配置器

二级空间配置器

9.vector和list的区别和应用？怎么找到倒数第二个元素？

10.STL中vector删除元素，迭代器怎么变化？为什么是两倍扩容？释放空间？

11.容器内删除一个元素

12.STL迭代器如何实现

13.map，set的实现？红黑树怎么能够同时实现这两种容器？为什么用红黑树？

14.如何在共享内存上使用STL标准库？

15.map插入方式有几种？

16. unordered_map(hash_map)和map的区别，hash_map如何解决冲突以及扩容？

17.vector越界访问下标，map越界访问下标？vector删除元素时会不会释放空间？

18.map中的[]和find的区别

19.STL中list和deque的区别

20.STL中的allocator、deallocator

21.STL中hash_table扩容发生什么？

22.常见容器性质总结

23.vector的增加删除都是怎么做的？为什么是1.5或者是2倍？

24.STL每种容器对应的迭代器

25.STL迭代器失效的情况？

26.vector实现

27.slist实现

28.list实现

29.deque的实现

30.STL中stack和queue的实现

31.Heap的实现

32.priority_queue实现

33.set的实现

34.map的实现
35.set/map，multimap/multiset区别
36.红黑树概念
37.unordered_map和map的区别和应用场景
38.hash_table中解决冲突的方法
其余问题
1.C++的多态如何实现
2.为什么析构函数一般写成虚函数
3、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？
4.基类的虚函数表放在内存的什么区？虚函数表vptr的初始化时间？
5.模板函数和模板类的特例化
6.构造函数、析构函数、虚函数可否声明为内联函数?
7.C++模板是什么，底层怎么实现？
8.构造函数为什么不能是虚函数？析构函数为什么要虚函数？
9.析构函数的作用？怎么起作用？
10.构造函数和析构函数中可以调用虚函数么？
11.构造析构的执行顺序？构造函数和拷贝构造的内部都干了啥？
12.构造函数析构函数可否抛出异常
13.构造函数一般不定义为虚函数的原因
14.类在什么时候析构？
15.构造函数的几种关键字
16.构造函数、拷贝构造函数和赋值操作符的区别
17.拷贝构造和赋值运算符重载的区别？
18.虚拟继承
19.什么情况下自动生成默认构造？
20.抽象基类为什么不能创建对象？
21.模板类和模板函数的区别？
22.多继承的优点？
23.模板和实现能够不写在一个文件里？为什么？
24.将字符串hello world从开始到打印到屏幕上的全过程？
25.为什么拷贝构造函数必须传引用不能传值？
26.静态函数能定义为虚函数吗？常函数呢？说说你的理解
27.虚函数的代价
28.说一下移动构造函数
29.那什么时候需要合成拷贝构造函数呢？
30.构造函数的执行顺序
31.哪些函数不能是虚函数？把你知道的都说一说
32.什么是纯虚函数，与虚函数的区别

一、C/C++基础语法

1.main之前和之后

之前：主要是静态和全局变量

1.设置栈指针

2.`.data`的内容，static和global变量

3.`.bss`的内容，全局变量赋初值

4.全局对象初始化

5.传参数argc argv 6.`__attribute__((constructor))`

之后：

1.全局对象析构

2.`atexit`注册一个函数

3.`__attribute__((destructor))`

2.结构体内存对齐

3.指针和引用的区别

指针	引用
是变量，存储地址	原变量的别名
可以有多级	只有一级
可以为空	不能为null，定义时要初始化
初始化后能改变指向	不能改变指向
sizeof是指针本身的大小	sizeof是变量本身的大小

指针	引用
改变指针的形参不会影响实参	会改变
是具体变量	本质是指针占4字节

4.何时用指针，何时用引用

1. 返回函数内局部变量的内存时用指针，用完要释放
2. 对栈空间大小敏感时用引用，不需要开辟空间
3. 类对象作为参数传递，用引用，标准方式

5.堆和栈的区别

1. 申请方式：堆是自己
2. 申请大小的限制：堆不连续，大小可以调整（1-4G）

   栈（4M)，堆向高地址方向增长，栈是向低地址方向
3. 申请效率：自己分配，速度慢有碎片

堆比较慢，要找合适大小的内存，获取堆的内容要访问两次，一次是指针，另一次是指针指向的地址。

6.区别指针类型

```
int *p[10]  // 存放了10个指针的指针数组
int (*p)[10]    // 存放了10个int的数组的数组指针
int *p(int)     // 函数声明 参数是int 返回值是int *
int (*p)(int)   // 函数指针 指针指向的函数是int类型的参数，返回值也是类型
```

7.new/delete 和 malloc/free的异同

1.new和delete如何实现的

- new的实现过程是：首先调用名为**operator new**的标准库函数，分配足够大的原始为类型化的内存，以保存指定类型的一个对象；接下来运行该类型的一个构造函数，用指定初始化构造对象；最后返回指向新分配并构造后的的对象的指针
- delete的实现过程：对指针指向的对象运行适当的析构函数；然后通过调用名为**operator delete**的标准库函数释放该对象所用内存

new：内存分配+初始化

delete：析构+内存释放

大片区域就不行了，没有默认构造的类不能分配动态数组 。`allocator`分配

2.异同

相同：用于内存的动态申请和释放

不同：

1. 前者C++，后者C；
2. new自动计算分配的大小，malloc要手工计算；
3. new是类型安全的，malloc不是。
4. 后者没有2中的调用
5. 后者要库文件，前者不用
6. new封装了malloc，直接free不会报错，但是只会释放内存，不会析构对象。

更精炼一点：功能，返回值

NEW/DELETE	MALLOC/FREE
是运算符，支持重载	标准库函数
new和delete除了分配回收功能外，还会调用构造函数和析构函数。	malloc仅仅分配内存空间，free仅仅回收空间，不具备调用构造函数和析构函数功能，用malloc分配空间存储类的对象存在风险
返回的是具体类型指针	返回的是void类型指针（必须进行类型转换

8.free之后怎么办

`ptmalloc`使用双链表保存起来，当用户下一次申请内存的时候，会尝试从这些内存中寻找合适的返回。这样就避免了频繁的系统调用，占用过多的系统资源。同时`ptmalloc`也会尝试对小块内存进行合并，避免过多的内存碎片。

`ptmalloc`就是管理的free之后的遗留问题的。

9.宏定义和函数以及typedef的区别

宏定义	函数	TYPEDEF
预处理阶段完成替换	调用的时候跳转	typedef是编译的一部分
属于在结构中插入代码，没有返回值	要返回值	
参数没有类型，不进行类型检查	要检查	检查数据类型
不要在最后加分号		是语句，要加分号标识结束
宏主要用于定义常量及书写复杂的内容		用于定义类型别名

注意对指针的操作，typedef char * p_char和#define p_char char *区别巨大

10.变量的声明和定义

声明提供位置和类型，不分配内存空间；定义需要

可在多处声明，但只能在一处定义。

11.strlen和sizeof的区别

STRLEN	SIZEOF
字符串的库函数	运算符
只能是字符指针且结尾是'\0'的字符串	任何数据的类型和数据
	编译时确定，不能用来得到动态分配存储空间的大小

```
const char* str = "name";
sizeof(str); // 取的是指针str的长度，是8
strlen(str); // 取的是这个字符串的长度，不包含结尾的 \0。大小是4
```

12.一个指针占多少字节？

64位的编译环境下 8字节

32位 4字节

13.指针常量和常量指针区别

const开头就是指针常量， *const是常量指针

1. 常量指针（const pointer）：指针的指向不能改（里面存放的地址不能修改），指针指向的值可以改 ,顶层const

```
int *const cur = &num
```

2. 指针常量(pointer to const):即指向常量的指针，指针指向的值不能改（指针的值可以通过别的方式修改，但是不能通过这个指针修改），指针的指向可以改 ,底层const

```
const int *p = &a
```

顶层const：指针本身是个常量，常被忽略
底层const：指针所指向的对象是个常量
*在const前，说明这个指针是个常量，是常量指针，是顶层的const

14.a和&a有什么区别

15.C++和Python的区别

是脚本语言，解释执行，编译语言要编译之后再特定平台运行的。

不用预先定义，数据类型少，缩进来区分代码块

解释型：源代码—>中间代码—>机器语言

不同平台对编译器影响较大。
如：

- （1）16位系统下int是2个字节（16位），而32位系统下int占4个字节（32位）；
- （2）32位系统下long类型占4字节，而64位系统下long类型占8个字节；

16.C和C++的区别

string stdio malloc/free try/catch setjump longjump

C++可以重载，C语言不允许

C++中，除了值和指针之外，新增了引用

C++相对与C增加了一些关键字，如：bool、using、dynamic_cast、namespace等等

17.C++与JAVA的区别

C++	JAVA
	完全面向对象，可移植性强，JVM中得到结果
类	接口技术省却了在实现和维护上的复杂性
	引入了真正的数组
	不考虑内存分配和回收问题，框架多

18.C++中的struct和class

同：都有成员函数，公有和私有部分；能用class的一定能用struct

异：默认struct是public，class是private；继承也是如此；定义模板的话不能用template<struct T>

19.define宏定义和const的区别

编译阶段：define在预处理阶段，const是编译运行时

安全：define替换，不检查；const有数据类型，可以进行安全检查

内存占用：define在内存中有多份相同备份，const 运行中只有一份，可移植性常量折叠，复杂表达式直接放入常量表

20.C++中的const和static的区别

	考虑类	不考虑类
static	成员变量：只与类有关，不和对象关联。成员函数：无this指针，不能被声明为const，虚函数和volatile	隐藏：加了static的全局变量和函数就只能在该文件的编译模块中使用。默认初始化为0。静态变量在函数内定义，始终存在，函数退出后存在但不可用
const	成员变量：只能通过构造函数初始化，必须有构造函数。成员函数：const对象不可以调用非const成员函数，不可以改变非mutable数据的值	常量在定义时初始化，不可修改。const形参可以接受const和非const的实参。$intfun(consti)$这个函数可以传入 int i 和const int i

21.顶层const和底层const

顶层：const修饰的变量本身是一个常量，指的是指针本身

底层：指针指向的对象是一个常量，指的是变量

- 执行对象拷贝时有限制，常量的底层const不能赋值给非常量的底层const
- 使用命名的强制类型转换函数const_cast时，只能改变运算对象的底层const

22.final和override关键字

重写和终止继承

OVERRIDE	它指定了子类的这个虚函数是重写的父类的，如果你名字不小心打错了的话，编译器是不会编译通过的

OVERRIDE	它指定了子类的这个虚函数是重写的父类的，如果你名字不小心打错了的话，编译器是不会编译通过的
FINAL	当不希望某个类被继承，或不希望某个虚函数被重写，可以在类名和虚函数后添加**final**关键字，添加**final**关键字后被继承或重写，编译器会报错

23.拷贝初始化和直接初始化

- 直接初始化直接调用与实参匹配的构造函数，拷贝初始化总是调用拷贝构造函数。拷贝初始化首先使用指定构造函数创建一个临时对象，然后用拷贝构造函数将那个临时对象拷贝到正在创建的对象
- 为了提高效率，允许编译器跳过创建临时对象这一步，直接调用构造函数构造要创建的对象，这样就完全等价于直接初始化了。要辨别两种情况。
    - 当拷贝构造函数为private时：语句3和语句4在编译时会报错
    - 使用explicit修饰构造函数时：如果构造函数存在隐式转换，编译时会报错

```
string str1("I am a string");//语句1 直接初始化
string str2(str1);//语句2 直接初始化，str1是已经存在的对象，直接调用拷贝构造函数对str2进行初始化
string str3 = "I am a string";//语句3 拷贝初始化，先为字符串”I am a string“创建临时对象，再把临时对象作为参数，使用拷贝构造函数构造str3
string str4 = str1;//语句4 拷贝初始化，这里相当于隐式调用拷贝构造函数，而不是调用赋值运算符函数
```

24.初始化和赋值的区别

- 对于简单类型来说，初始化和赋值没什么区别
- 复杂数据类型，赋值的话会涉及到等号的重载，这个过程中会对值产生影响。

```
class A{
public:
    int num1;
    int num2;
public:
    A(int a=0, int b=0):num1(a),num2(b){};
    A(const A& a){};
```

```
    //重载 = 号操作符函数
    A& operator=(const A& a){
        num1 = a.num1 + 1;
        num2 = a.num2 + 1;
        return *this;
    };
};
int main(){
    A a(1,1);
    A a1 = a; //拷贝初始化操作，调用拷贝构造函数
    A b;
    b = a;//赋值操作，对象a中，num1 = 1，num2 = 1；对象b中，num1 = 2，num2 = 2
    return 0;
}
```

25.extern "C"的用法

为了能够正确的让C++调用C的代码，告诉编译器是是C写的。哪些情况下使用

- C++代码调用C
- C++头文件中使用
- 多人协作

在C语言中，外部调用是指定为extern类型。

26.野指针和空悬指针

都是指向无效内存区域，访问行为导致未定义行为

- 野指针：没被初始化过的指针，因此初始化的时候为nullptr
- 悬空指针：指针最初指向的内存已经被释放了的一种指针。delete之后，没有设置为nullptr。引入智能指针避免之。

27.C和C++的类型安全

28.C++中的重载、重写（覆盖）和隐藏的区别

重载：overload，函数名相同，参数类型和数目不同，返回值不同

重写：override，在派生类中覆盖基类的同名函数，基类函数必须是虚函数。返回值，参数类型，参数数量都要完全一致

隐藏：hide，派生类中的函数屏蔽了基类中的同名函数。

两个函数参数相同，但是基类函数不是虚函数；

两个函数参数不同，无论基类函数是不是虚函数，都会被隐藏

29. C++有哪几种构造函数

三/五法则：三个基本操作：拷贝构造，拷贝赋值和析构。新标准引入了移动构造和移动赋值。

- 默认构造函数
- 初始化构造函数（有参数）
- 拷贝构造函数
- 移动构造函数（move和右值引用）
- 委托构造函数
- 转换构造函数

```
#include <iostream>
class Base
{
public:
    int value1;
    int value2;
    Base()    //目标构造函数
    {
        value1 = 1;
```

```
    }
    Base(int value) : Base()  //委托构造函数
    { // 委托 Base() 构造函数
        value2 = value;
    }
};
void EntrustedConstruction()
{
    Base b(2);  //首先调用Base(int value) : Base() 毫无疑问
    //然后会走到base()中，先给value1复制，然后走到Base(int value) : Base() ，给value2赋值
    std::cout << b.value1 << std::endl;
    std::cout << b.value2 << std::endl;
----------------
版权声明：本文为CSDN博主「CodeBowl」的原创文章，遵循CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。
原文链接：https://blog.csdn.net/CodeBowl/article/details/120102112
```

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

class Student{
public:
    Student(){//默认构造函数，没有参数
        this->age = 20;
        this->num = 1000;
    };
    Student(int a, int n):age(a), num(n){}; //初始化构造函数，有参数和参数列表
    Student(const Student& s){//拷贝构造函数，这里与编译器生成的一致
        this->age = s.age;
        this->num = s.num;
    };
    Student(int r){   //转换构造函数,形参是其他类型变量，且只有一个形参
        this->age = r;
        this->num = 1002;
    };
    ~Student(){}
```

```
public:
    int age;
    int num;
};
```

30.浅拷贝和深拷贝的区别

浅拷贝

浅拷贝只是拷贝一个指针，并没有新开辟一个地址，拷贝的指针和原来的指针指向同一块地址，如果原来的指针所指向的资源释放了，那么再释放浅拷贝的指针的资源就会出现错误。

深拷贝

深拷贝不仅拷贝值，还开辟出一块新的空间用来存放新的值，即使原先的对象被析构掉，释放内存了也不会影响到深拷贝得到的值。在自己实现拷贝赋值的时候，如果有指针变量的话是需要自己实现深拷贝的。

31.内联函数和宏定义的区别

使用宏定义的地方都可以使用inline函数

内联函数不会 执行参数检查

32.public，protected和private的访问继承

一、访问权限

外部 派生类 内部

二、继承权限

1. 派生类继承自基类的成员权限有四种状态：public、protected、private、不可见
2. 派生类对基类成员的访问权限取决于两点：一、继承方式；二、基类成员在基类中的访问权限
3. 派生类对基类成员的访问权限是取以上两点中的更小的访问范围（除了 private 的继承方式遇到 private 成员是不可见外

- public 继承 + private 成员 => private
- private 继承 + protected 成员 => private
- private 继承 + private 成员 => 不可见

33.如何用代码判断大小端存储

大端存储：字数据的高字节在低地址中

小端存储：字数据的低字节存储在低地址中

所以在**Socket**编程中，往往需要将操作系统所用的小端存储的**IP**地址转换为大端存储，这样才能进行网络传输

方式一：使用强制类型转换

方式二：巧用**union**联合体

34.volatile，mutable和explict用法

1.volatile

volatile定义变量的值是易变的，每次用到这个变量的值的时候都要去重新读取这个变量的值，而不是读寄存器内的备份。多线程中被几个任务共享的变量需要定义为**volatile**类型。

2.mutable

在C++中，mutable也是为了突破const的限制而设置的。被mutable修饰的变量，将永远处于可变的状态

我们需要在**const**函数里面修改一些跟类状态无关的数据成员，那么这个函数就应该被**mutable**来修饰，并且放在函数后后面关键字位置。

3.explict

explicit关键字用来修饰类的构造函数，被修饰的构造函数的类，不能发生相应的隐式类型转换，只能以显示的方式进行类型转换，

35.什么情况下会调用拷贝构造函数

1. 用类的一个对象去初始化另一个对象时
2. 当函数的形参是类的对象时（也就是值传递时），如果是引。用传递则不会调用。调用时产生临时对象，再赋值、
3. 当函数的返回值是类的对象或引用时，返回时会产生临时对象。

36.C++中有几种类型的new

三个典型的使用方法： plain new， nothrow new和 placement new

plain new 在空间分配失败的情况下，抛出异常std::bad_alloc而不是返回NULL

nothrow new 在空间分配失败的情况下是不抛出异常，而是返回NULL

placement new允许在一块已经分配成功的内存上重新构造对象或对象数组。placement new不用担心内存分配失败，因为它根本不分配内存，它做的唯一一件事情就是调用对象的构造函数

- palcement new的主要用途就是反复使用一块较大的动态分配的内存来构造不同类型的对象或者他们的数组
- placement new构造起来的对象数组，要显式的调用他们的析构函数来销毁（析构函数并不释放对象的内存），千万不要使用delete，这是因为placement new构造起来的对象或数组大小并不一定等于原来分配的内存大小，使用delete会造成内存泄漏或者之后释放内存时出现运行时错误。

37.C++的异常处理的方法

常见的异常：数组下标越界，除以0，动态分配的空间不足

1. try，throw和catch关键字（try包含，throw抛出，catch捕获）
2. 预先知道可能的异常，提前指出抛出异常的列表
3. 异常类exception可以派生出很多

   out_of_range 越界 bad_alloc无足够内存

   bad_cast多态对象的转换 bad_typeid多态的指针

38.static的用法和作用

作用1	**隐藏，所有编译文件中的全局变量和函数，加了就会隐藏**
作用2	保持变量内容的持久，静态存储区的变量在程序开始运行时完成初始化，全局变量和static变量就在静态存储区
作用3	默认初始化为0
作用4	1.维持上次的值2.static的全局变量只能被模块内访问和调用3.类内的static只有一份，成员函数不接受this指针，不能被virtual修饰，不属于任何一个对象

39.形参和实参的区别

1.形参在调用时才会分配内存，结束后释放，只在函数内部有效

2.实参在调用函数之前就已经赋值了，然后再传给形参

3.只能将实参传给形参，调用过程中形参的变化不会影响实参

4.如果不是指针类型，在内存中是两个变量和存储空间

40.值传递，指针传递，引用传递

1.传值只把数值传过去，要拷贝一份过去

2.传指针传递拷贝的是地址的值

3.传引用拷贝的是地址的别名

23更好，3最好逻辑紧凑清晰

41.静态变量的初始化？

全局区 （全局变量，静态变量，常量）堆区（程序员的分配） 栈区（局部变量 局部常量）

1.一次初始化多次复制，主程序之前分配内存

2.静态变量在全局区，静态局部变量在C++中只有用的时候再初始化，进行构造和析构。

42.const关键字的作用

1.字面意思，阻止变量被改变，声明的时候必须初始化

2.指针可以指向和本身都设置为const

3.函数声明中设置为const，让它变为输入参数，不改变值

4.类的const成员函数是常函数，不能修改成员变量，常对象访问常函数，因为不是常函数的话编译器不允许作为const对象调用

5.非const成员函数不能访问const对象的数据成员

6.const_cast将const转换成非const

7.只有传指针和传引用的可以用const重载，传值的不行

43.类的继承

1.类的关系 has-a use-a

2.继承的相关概念 基类派生类 子类父类

3.继承的特点：子类对象可以当做父类对象是用

4.访问控制：public protected private

5.构造析构

44.汇编层解释引用

传引用把地址移动到引用变量的位置

45.delete， delete[] allocator的作用

1.动态数组管理new一个数组，[]中是一个整数，但不一定是常量整数，普通数组必须是一个常量整数

2.new返回的是元素类型的指针

3.delete[] 数组中的元素按逆序销毁

4.new=内存分配+对象构造，allocator申请一部分内存，不初始化对象，需要的时候才初始化

46.new和delete原理，delete怎么知道释放内存大小

new表达式调用一个名为operator new(operator new[])函数，然后就是构造和初始化，

需要在 new [] 一个对象数组时，需要保存数组的维度，C++ 的做法是在分配数组空间时多分配了 4 个字节的大小，专门保存数组的大小，在 delete [] 时就可以取出这个保存的数，就知道了需要调用析构函数多少次了。

47.malloc申请的存储空间能用delete释放么

理论上可以

new 和delete会自动进行类型检查和大小，malloc/free不能执行构造函数与析构函数，所以动态对象它是不行的。

48.malloc和free实现原理（太底层）

49.malloc realloc calloc的区别

realloc省去人为空间计算，初始化为0

realloc给动态内存分配额外的空间

50.类成员初始化？构造函数执行程序？成员初始化列表更快？

1.赋值初始化和列表初始化。前者在函数体中初始化，所有数据成员分配空间后才进行；后者在之前就初始化了。

2.构造函数的执行程序如下

虚拟基类的构造函数（按照继承顺序执行构造）

基类的构造函数（普通基类顺序执行构造）

类类型的成员对象的构造

派生类自己的构造

3.方法一再构造函数中赋值，方法二纯粹初始化，方法一会产生临时对象，比较慢

初始化列表只赋值，调用有参构造；

构造函数中赋值的话默认构造+赋值

成员初始化列表和列表初始化（vector的初始化）

51.哪些情况必须列表初始化？作用是什么？

初始化一个引用成员；常量成员；调用的基类构造函数有一组参数；调用一个成员类的构造函数有一组参数

做了什么？是根据成员声明决定的，不是顺序

52.char*和string的区别，如何实现

string继承自basic_string，其实是对char* 进行封装，包含了char*数组，容量长度等属性。

string可以动态扩展，每次扩展申请原空间两倍的空间，再将原字符拷贝过去，再加上新增内容。

53.什么是内存泄漏？检测与避免？

说的一般是堆内存泄露，必须在new alloc realloc之后调用free和delete释放，否则不能再次利用，就是内存泄漏

避免方法：

计数法，新建就记录+1

将基类的析构函数声明为虚函数

（通过基类指针销毁子类时，调用静态绑定的析构函数，也就是基类的构造函数，所以只能销毁基类的元素；而且子类不使用动态资源或者其他自定义析构行为的时候，可以不写虚析构函数，提升效率）

对象数组用delete[]

new delete一定要配对

54.对象复用的了解，零拷贝的了解

对象复用：是一种设计模式Flyweight享元模式，将对象存储到对象池中重复利用，避免多次创建重复对象的开销，节省资源

零拷贝：避免CPU将数据从一块存储拷贝到另一块存储的技术

减少数据拷贝和共享总线操作的次数。emplace_back原地构造，push_back先调用拷贝构造和转移构造。

55.介绍面向对象的三大特性，举例说明

继承封装和多态

继承：某个类型对象获得另一个类型对象的属性和方法

1.实现继承，直接用

2.接口继承：子类重写

3.可视继承：子类使用基类的外观和实现代码的能力

封装：数据和代码捆绑在一起，避免外界干扰和不确定性访问

把客观事物封装成抽象的；类内通过自己设定的方法访问

多态：同一个实物表现出不同事物的能力，向不同对象发送同一个消息，产生的行为不同（重载是编译时多态，虚函数是运行时多态）

允许将子类类型的指针赋值给父类类型的指针，因为子类一定有父类的东西

两种实现方式：重载（overload）和重写（override）

56.四种强制类型转换

REINTERPRET_CAST	**REINTERPRET_CAST<TYPE-ID> (EXPRESSION)TYPE-ID 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。**
const_cast	const_cast<type_id> (expression)该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type_id和expression的类型是一样的。常量的指针和引用转换为非常量的。const int* -> int *
static_cast	没有类型检查保证转换的安全性。父类子类之间，子转父（上行）安全，下行不安全，基本数据类型转换int->char 空指针转换为目标类型的指针，任何表达式转为void

REINTERPRET_CAST	**REINTERPRET_CAST<TYPE-ID> (EXPRESSION)TYPE-ID 必须是一个指针、引用、算术类型、函数指针或者成员指针。它可以用于类型之间进行强制转换。**
dynamic_cast	有类型检查，基类转派生安全，子转父不安全。type_id必须是类的指针，类的引用或者void*，下行安全，**如果实际不安全，就返回空指针**。类之间的转换

57.C++函数调用的压栈过程

类似于中断

58.C++中的一类错误 coredump，怎么调试？

coredump是程序由于异常或者bug在运行时异常退出或者终止，在一定的条件下生成的一个叫做core的文件，这个core文件会记录程序在运行时的内存，寄存器状态，内存指针和函数堆栈信息等等。对这个文件进行分析可以定位到程序异常的时候对应的堆栈调用信息。

gdb调试编译生成好的文件。

59.移动构造函数？

新标准引入了移动构造和移动赋值

1.用a初始化b，a没了，那就直接用a来原地创建b，省空间

2.移动构造的指针是浅层赋值，避免释放第一个指针的空间。

3.移动构造是右值引用，move将左值变右值。

60.怎么获得结构成员相对于结构开头的字节偏移量

stddef.h的offsetof宏

61.静态类型和动态类型？动态绑定和静态绑定？

- 静态类型：对象在声明时采用的类型，在编译期既已确定；
- 动态类型：通常是指一个指针或引用目前所指对象的类型，是在运行期决定的；
- 静态绑定：绑定的是静态类型，所对应的函数或属性依赖于对象的静态类型，发生在编译期；
- 动态绑定：绑定的是动态类型，所对应的函数或属性依赖于对象的动态类型，发生在运行期；

在继承体系中只有虚函数使用的是动态绑定，其他的全部是静态绑定

62.引用能否实现动态绑定？

可以。访问虚函数可以实现动态绑定，所以只能调用虚函数。

虚函数才具有动态绑定

63.全局变量和局部变量

生命周期：全局变量跟着主程序走

使用方式不同：全局变量在程序的各个部分都能用到，局部变量在堆栈区

操作系统和编译器能够通过位置区分两者。

64.指针加减计算要注意什么？

移动一位表示跨越的内存间隔是指针类型的长度。

65.怎么判断两个浮点数相等？

要根据精度来比较。区绝对值。

66.方法调用的原理

中断的原理

67.C++的传指针和传引用的区别和原理？

简言之：值传递只复制了值，传指针虽然是指针，但是复制以后指针变量的地址，指针变量本身指向的值还是原来的值，传引用是将实参的地址传进来了。

1) 指针参数传递本质上是值传递，它所传递的是一个地址值。

值传递过程中，被调函数的形式参数作为被调函数的局部变量处理，会在栈中开辟内存空间以存放由主调函数传递进来的实参值，从而形成了实参的一个副本（替身）。

值传递的特点是，被调函数对形式参数的任何操作都是作为局部变量进行的，不会影响主调函数的实参变量的值（形参指针变了，实参指针不会变）。

2) 引用参数传递过程中，被调函数的形式参数也作为局部变量在栈中开辟了内存空间，但是这时存放的是由主调函数放进来的实参变量的地址。

3) 引用传递和指针传递是不同的，虽然他们都是在被调函数栈空间上的一个局部变量，但是任何对于引用参数的处理都会通过一个间接寻址的方式操作到主调函数中的相关变量。

而对于指针传递的参数，如果改变被调函数中的指针地址，它将应用不到主调函数的相关变量。如果想通过指针参数传递来改变主调函数中的相关变量（地址），那就得使用指向指针的指针或者指针引用

4) 从编译的角度来讲，程序在编译时分别将指针和引用添加到符号表上，符号表中记录的是变量名及变量所对应地址。

指针变量在符号表上对应的地址值为指针变量的地址值，而引用在符号表上对应的地址值为引用对象的地址值（与实参名字不同，地址相同）。

符号表生成之后就不会再改，因此指针可以改变其指向的对象（指针变量中的值可以改），而引用对象则不能修改。

68.类如何实现只能静态分配和只能动态分配？

栈上的是静态，堆上的是动态

new建立在堆上，只要限制它就能建立在栈上

将new设置为私有就可以建立在栈上，实现静态分配。

把构造和析构设为protected，用子类就能动态创建。

69.如果将某个类用作基类，为什么必须定义而非声明？

派生类中包含并且可以使用从基类继承的成员，派生类必须知道他们是什么，所以要定义。

70.继承中对象如何转换？指针和引用如何转换？

向上类型转换，安全

向下类型转换，不会自动进行，一个父类多个子类不知道转哪一个，要加动态类型识别技术，RTTI(run-time type identification)，用dynamic_cast向下转。

71.C++的组合？它和继承相比有什么优缺点？

	优点	缺点
继承	重写父类实现扩展	父类细节子类可见；无法在运行期间改变继承的方法；父类修改子类就要修改，高耦合
组合	当前对象通过所包含的对象调用，细节不可见；两者低耦合，修改包含对象的代码不影响当前对象；在运行时当前对象动态绑定所包含的对象，通过set方法赋值	容易产生过多的对象；为了组合多个对象，要仔细对接口进行定义

72.函数指针

指向特殊的数据类型，指向函数的地址

int (*pf)(int a, int b),通过函数指针指向函数的代码

两种复制方法： 函数名=指针名； &函数名=指针名

73.说一下你理解的内存对齐以及原因

分配内存的顺序是按照声明的顺序；

每个变量相对于起始位置的偏移量必须是该变量类型大小的整数倍，如果过不是就空出来，直到满足这个条件；

最后整个结构体的大小是里面边疆类型最大值的整数倍。

添加了#pragma pack(n)后规则就变成了下面这样：

偏移量要是n和当前变量大小中较小值的整数倍；

整体大小是n和最大变量大小中较小值的整数倍；

n=1，2，4，8...

对齐的作用和原因：各个硬件平台对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。其他平台可能没有这种情况，但是最常见的是如果不按照适合其平台要求对数据存放进行对齐，会在存取效率上带来损失。比如有些平台每次读都是从偶地址开始，如果一个int型（假设为32位系统）如果存放在偶地址开始的地方，那么一个读周期就可以读出，而如果存放在奇地址开始的地方，就可能会需要2个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该int数据。显然在读取效率上下降很多。这也是空间和时间的博弈。

74.结构体变量比较是否相等

指针直接比较，元素的话一个一个比

75.函数调用过程栈的变化，返回值和参数变量哪个先入栈？

先定义的变量先入栈，后定义的后入栈。

对于形参来讲，从右向左依次把被调函数所需要的参数压入栈中。

函数调用过程中，第一个进栈的是（主函数中的） 调用处的下一条指令（即函数调用语句的下一条可执行语句） 的地址；然后是函数的各个参数，而在大多数C/C++编译器中，在函数调用的过程中，函数的参数是 由右向左入栈的；然后是函数内部的 局部变量（注意static变量是不入栈的）；在函数调用结束（函数运行结束）后，局部变量最先出栈，然后是参数，最后栈顶指针指向最开始存的指令地址，程序由该点继续运行。

76.define,const,typedef,inline的使用方法和区别？

CONST	#DEFINE
定义的常量是变量+类型	只是个常数 无类型
编译链接过程中	预处理阶段
有数据类型判断	简单替换无判断
不能重定义	可以用#undef取消再定义
	防止头文件重复包含

DEFINE	INLINE
关键字	函数
预处理替换	编译阶段替换
	有类型检查，更安全

#DEFINE	TYPEDEF
预处理阶段	编译阶段
取别名，定义常量变量	定义函数类型的别名，与平台无关的数据类型
无作用域限制	有自己的作用域

77.printf的实现原理

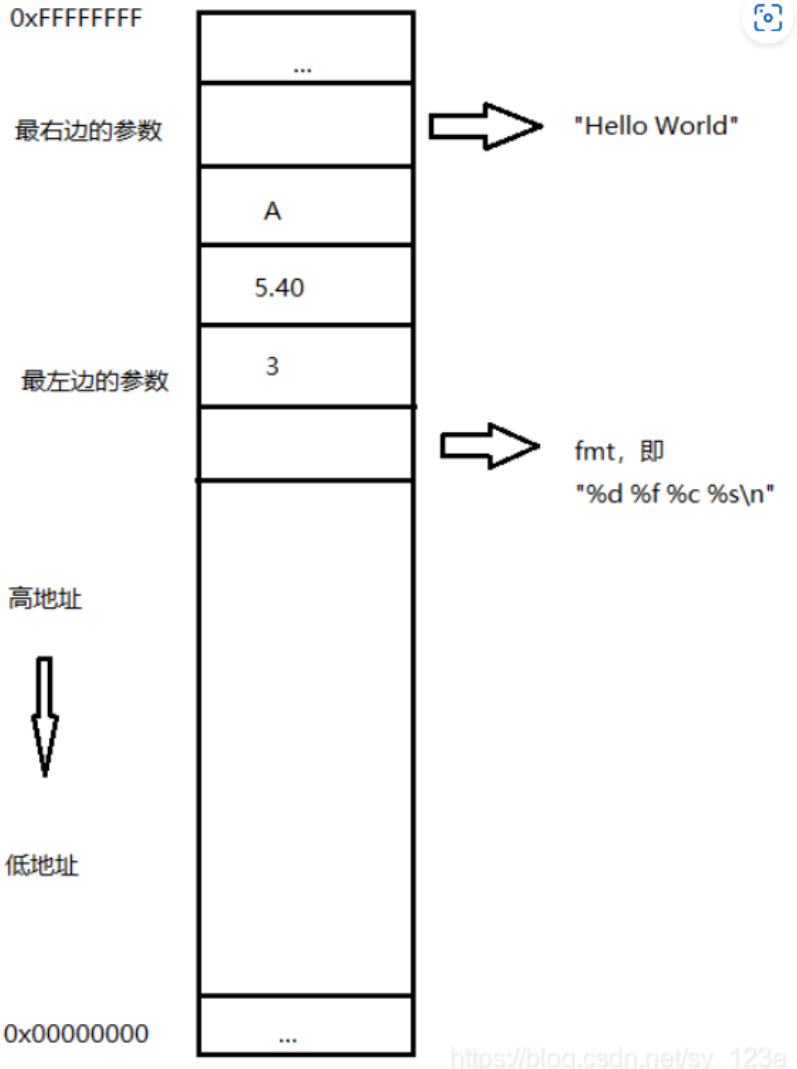
参数传入堆栈，从右向左压栈，栈的生长是从高地址指向低地址。

对于c语言，它的调用规则遵循_cdedl调用规则。 在_cdedl规则中：

1. 参数从右到左依次入栈
2. 调用者负责清理堆栈
3. 参数的数量类型不会导致编译阶段的错误

printf的声明： int _cdecl printf(const char* format, …); // _cdecl是C和C + + 程序的缺省调用方式

栈由高地址向低地址生长，又参数从右到左依次入栈，所以栈的高地址是printf最右边的参数。以printf("%d %f %c %s\n", 3, 5.40, A, "hello world");为例，其栈的结构如下：



printf的第一个被找到的参数就是那个字符指针，就是被双引号括起来的那一部分，函数通过判断字符串里控制参数的个数来判断参数个数及数据类型，通过这些就可**算出数据需要的堆栈指针的偏移量**了，下面给出printf("%d,%d",a,b);（其中a、b都是int型的）的汇编代码.

78、为什么模板类一般都是放在一个h文件中

1.模板的定义很特殊，意味着不能为其分配存储空间，可以去掉多重定义

2.分离式编译的情况下，cpp文件之间互相不知道存在，在没模板的时候可以，但是有模板的时候就不行了。编译器看到模板声明，但不能实例化，就只能创建一个具有外部链接的符号并期望外部链接确定符号的地址，但是在实现该模板的类中如果过没有用到实例，就不会实例化，整个工程的obj中就找不到一行模板实例的二进制代码。

https://blog.csdn.net/lijiayu2015/article/details/52650790说到底就是因为模板如果不实例化就不生成二进制代码

79.C++类成员的访问权限和继承权限问题

32.public，protected和private的访问继承

① 若继承方式是public，基类成员在派生类中的访问权限保持不变，也就是说，基类中的成员访问权限，在派生类中仍然保持原来的访问权限；

② 若继承方式是private，基类所有成员在派生类中的访问权限都会变为私有(private)权限；

③ 若继承方式是protected，基类的共有成员和保护成员在派生类中的访问权限都会变为保护(protected)权限，私有成员在派生类中的访问权限仍然是私有(private)权限。

80.cout和printf

cout<< 支持多种数据 有重载 是一个对象

先放入缓冲区，再输出到桌面

```
cout < < "abc " < <endl;
或cout < < "abc\n "; cout < <flush; 这两个才是一样的
```

flush立即强迫缓冲输出

printf是行缓冲输出，不是无缓冲输出 是一个函数

81.重载运算符？函数匹配的顺序和原则？

通过重载实现多态，重载已有的运算符，运算的优先级和结合律与内置类型一样，运算符操作数个数也不能改变。

全局函数一般是两个参数，类内函数一般是一个参数。

查名字；确定候选函数；寻找最佳匹配。

82.定义和声明的区别

变量：声明不会分配内存，定义会

函数：声明在头文件里，让编译器知道函数的存在，定义在源文件里，函数的实现过程。

83.全局变量和static变量的区别

非静态的全局变量：作用域是整个源程序，一个源程序由多个源文件组成时，非静态的全局变量在各个源文件中都是有效的。

静态全局变量：作用域只在定义该变量的源文件内有效，同一源程序的其他源文件中不能使用。

static全局变量只初始化一次，防止在其他文件单元中被引用。

全局变量就是静态存储方式。

static函数全局只有一份。

84.静态成员与普通成员的区别

	静态成员	普通成员
生命周期	类的开始和结束一直存在	类创建对象之后才开始存在
共享方式	全类共享	对象单独共享
定义位置	静态全局区	堆栈
初始化位置	类外初始化	类内初始化
默认实参	静态成员作为默认实参	

85.说一下你理解的ifdef endif代表着什么？

在一个大的软件工程里面，可能会有多个文件同时包含一个头文件，当这些文件编译链接成一个可执行文件上时，就会出现大量“重定义”错误。

```
#ifdef 标识符
程序段1
#else
程序段2
#endif
```

86.隐式类型转换，如何消除

1.变量来说可以方便使用，类的话，子类可以使用父类的类型返回

2.使用explicit，构造函数生命的时候加上，可以禁止隐式类型转换。

只对有一个实参的构造函数有效，多个实参的构造函数不能用于执行隐式转换。

87.C++如何处理多个异常？

自己不处理，抛出异常让函数的调用者直接或者间接的处理。

```
try  {  可能抛出异常的语句；（检查） try
{
可能抛出异常的语句；（检查）
}
```

```
catch（类型名[形参名]）//捕获特定类型的异常
{
//处理1；
}
catch（类型名[形参名]）//捕获特定类型的异常
{
//处理2；
}
catch（…）//捕获所有类型的异常
{
}
```

88.不使用额外空间交换两个数？

算术和异或

```
1)  算术

x = x + y;
 y = x - y;

x = x - y;

2)  异或

x = x^y;// 只能对int,char..
 y = x^y;
 x = x^y;
 x ^= y ^= x;
```

89.strcpy和memcpy

	STRCPY	MEMCPY
复制的内容不同	字符串	任意内容
复制的方法不同	不要指定长度，“\0”结束，容易溢出	根据第三个参数决定复制的长度
用途不同	字符串时使用	一般类型

90.程序执行int main(int argc, char *argv[])时的内存结构

argc个参数，每个参数都是以char类型输入的，存在数组argv[]中，所有的参数在指针char*指向的内存中，数组中元素的个数为argc个，第一个参数是程序的名称。

91.volatile关键字的作用

volatile 关键字是一种类型修饰符，用它声明的类型变量表示可以被某些编译器未知的因素更改

声明时语法：int volatile vInt; 当要求使用 volatile 声明的变量的值的时候，系统总是重新从它所在的内存读取数据，即使它前面的指令刚刚从该处读取过数据。而且读取的数据立刻被保存。

用在

1.中断服务程序中修改的供其他程序检测的变量要加volatile

2.多任务环境下各任务间共享的标志要加

3.存储器映射的硬件寄存器要加，因为每次的读写意义可能不同

92.如果有一个空类，会默认添加哪些函数？

```
1)  Empty(); // 缺省构造函数//
2)  Empty( const Empty& ); // 拷贝构造函数//
3)  ~Empty(); // 析构函数//
4)  Empty& operator=( const Empty& ); // 赋值运算符//
5、取地址操作符重载函数
6、const修饰的取地址操作符重载函数
```

缺省也叫默认构造/无参构造

93.C++中标准库是什么

分为标准函数库和面向对象类库。

前者继承自C语言，由通用的独立的不属于任何类的函数组成

后者是类及其相关函数的集合。

1. 前者：输入/输出 I/O、字符串和字符处理、数学、时间、日期和本地化、动态分配、其他、宽字符函数
2. 后者：标准的 C++ I/O 类、String 类、数值类、STL 容器类、STL 算法、STL 函数对象、STL 迭代器、STL 分配器、本地化库、异常处理类、杂项支持库

94.const char* 和 string的关系

string 是c++标准库里面其中一个，封装了对字符串的操作，实际操作过程我们可以用const char*给string类初始化

```
string转const char*
s.c_str()
const char* 转string
string s(c)
```

95.什么时候用指针当参数，什么时候用引用？

传引用和传指针可以提高程序运行速度，修改调用函数中的数据。

对象小	传值
对象是数组	只能用指针，const指针
较大结构	const指针/引用
类对象	const引用，是标准方式

如果修改函数中数据：

类对象引用，内置类型指针，对象是结构均可。

96.设计一个计算仅单个子类的对象个数

设置静态变量cnt作为计数器，类定义结束后初始化cnt；

默认构造拷贝构造和赋值构造中都要+1，析构中-1。

```
class Test
{
public:
 Test() :_a(0)
 {
  t_count++;
 }
 Test(Test& a)
 {
  t_count++;
 }
 ~Test()
 {
  t_count--;
 }
 static int GetCount()//静态成员函数
 {
 return t_count;
 }
 //int GetCount1()//普通成员函数
 //{
 // cout<<this<<endl;
 // return this->_a;
 //}
private:
 int _a;
 static int t_count;//静态成员变量    声明
};
int Test::t_count = 0;//静态成员变量在类外初始化，且不添加static关键字
void testCount()
{
 Test t1, t2;
 cout << Test::GetCount() << endl;
 //cout << t_count << endl;
```

```
}
int main()
{
 Test::GetCount();
 Test t1, t2;
 cout << Test::GetCount() << endl;//2
 //cout << t1.GetCount() << endl;//也可以通过对象访问
 testCount();//4
 cout << Test::GetCount() << endl;//2
 system("pause");
 return 0;
}
```

97.成员初始化列表在什么时候用到？调用过程是什么？

初始化引用成员变量时，一个const成员变量时，

继承一个基类的构造+一组参数：父类无构造但有参数，那子类必须初始化列表

调用一个成员类构造+一组参数的时候：成员类无构造但是有参数，所以也要初始化列表

生成顺序是按照声明决定的。

98.传引用的好处？

可修改，效率高。

限制：不能返回局部变量的引用；

不能返回函数内部new分配的内存的引用（返回的是如果是一个临时变量，那么指向的空间就无法释放，内存泄漏）；

可以返回类成员的引用，但最好是const。

99.说一说strcpy、sprintf与memcpy这三个函数的不同之处

89.strcpy和memcpy有说一些

	STRCPY	MEMCPY	SPRINTF
复制的内容不同	字符串	任意内容	源是多种数据类型，目的是字符串
复制的方法不同	不要指定长度，“\0”结束，容易溢出	根据第三个参数决定复制的长度	效率最低mem最高
用途不同	字符串时使用	一般类型	主要实现其他数据类型格式到字符串的转化

100. 引用作为函数的好处？

传指针和传引用效果一样，但是传引用避免了拷贝的开销，对象可能还会有构造函数的调用；传指针会重复使用指针变量名，易错且可阅读性差，引用更清晰。

101.数组和指针的区别？

连续开辟出来的区域。编译器用指针来管理数组，首地址+偏移量。

传过去的也是指针，但是数组的首地址是确定的，指针的源地址不是确定的。

102.阻止一个类被实例化的方法？

定义为抽象基类或者将构造函数声明为private

不允许类外部创建类对象，只能在类内部创建对象。

103.如何禁止程序自动生成拷贝构造？

1.为了阻止编译器默认生成拷贝构造函数和拷贝赋值函数，我们需要手动去重写这两个函数，某些情况下，为了避免调用拷贝构造函数和拷贝赋值函数，我们需要将他们设置成private，防止被调用。

2.类的成员函数和friend函数还是可以调用private函数，如果这个private函数只声明不定义，则会产生一个连接错误；

3.针对上述两种情况，我们可以定一个base类，在base类中将拷贝构造函数和拷贝赋值函数设置成private,那么派生类中编译器将不会自动生成这两个函数，且由于base类中该函数是私有的，因此，派生类将阻止编译器执行相关的操作。

104.Debug和Release的区别

Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合

1.调试版本，包含调试信息，所以容量比Release大很多，并且不进行任何优化（优化会使调试复杂化，因为源代码和生成的指令间关系会更复杂），便于程序员调试。Debug模式下生成两个文件，除了.exe或.dll文件外，还有一个.pdb文件，该文件记录了代码中断点等调试信息；

2.发布版本，不对源代码进行调试，编译时对应用程序的速度进行优化，使得程序在代码大小和运行速度上都是最优的。（调试信息可在单独的PDB文件中生成）。Release模式下生成一个文件.exe或.dll文件。

105.main函数的返回值

程序运行过程入口点main函数，main（）函数返回值类型必须是int，这样返回值才能传递给程序激活者（如操作系统）表示程序正常退出。

main（int args, char **argv） 参数的传递。参数的处理，一般会调用getopt（）函数处理，但实践中，这仅仅是一部分，不会经常用到的技能点。

106.strcpy和strncpy的区别

```
1 char* strcpy(char* strDest, const char* strSrc)
2 char *strncpy(char *dest, const char *src, size_t n)
```

strcpy和strncpy是早期C库函数，头文件string.h。现在已经发布对应safe版本，也就是strcpy_s, strncpy_s。

- strcpy函数: 如果参数 dest 所指的内存空间不够大，可能会造成缓冲溢出(buffer Overflow)的错误情况，在编写程序时请特别留意，或者用strncpy()来取代。

- strncpy函数：用来复制源字符串的前n个字符，src 和 dest 所指的内存区域不能重叠，且 dest 必须有足够的空间放置n个字符。

107.static_cast比C语言中的转换强在哪里

更加安全

更直接明显，能够一眼看出是什么类型转换为什么类型，容易找出程序中的错误；可清楚地辨别代码中每个显式的强制转；可读性更好，能体现程序员的意图

108.成员函数里memset(this,0,sizeof(*this))

直接就memset(this, 0, sizeof *this);将整个对象的内存全部置为0。

问题1：会破坏虚函数表

问题2：类中含有对象C++类型的对象，如果定义的对象已经初始化，会破坏内存。

109. 回调函数？作用？

发生某种事件的时候，系统或者其他函数将会自动调用它。

三个点：声明，定义和触发条件，符合条件自动调用。

本质就是一个通过函数指针调用的函数，当这个指针传给另一个函数的时候，这个指针用来指向那个函数，就是回调函数。

可以把调用者和被调用者分开。

110.什么是一致性哈希？

111.C++从代码到可执行程序经历了什么？

1.预编译，处理#开头的语句 4个#

展开宏定义#define，处理#ifdef等指令，处理#include指令，放到使用的位置，删除注释，保留#program once，

添加行号和文件标识，便于调试

2.编译

预编译后形成`xx.i`和`xx.ii`文件，进行词法，语法和语义分析，优化，生成`.s`文件

词法：字符序列分割成记号

语法：产生语法树，以表达式为节点

语义：子啊编译期能分析的语义

优化：源代码级别的优化过程

生成目标机器码（汇编表示），再优化，位移代替乘除，删除多余指令。

3.汇编

汇编代码->机器码，翻译，生成`xxx.o`和`xxx.obj`

4.链接

将不同源文件连接起来，形成可执行文件。

静态链接：编译成二进制文件，从库中复制函数和数据，组合起来形成可执行文件。

动态链接：把程序按照模块拆分成独立的部分，运行时链接成一个完整的程序，不像静态链接一样把所有的连接在一起。

每次都要链接，性能有损失。

112.友元函数和友元类

通过友元，一个不同函数或者另一个类中的成员函数可以访问类中的私有成员和保护成员。

友元类中，类的成员函数都能够访问当前类的私有和保护成员。

1.友元不能够被继承

2.友元是单向的

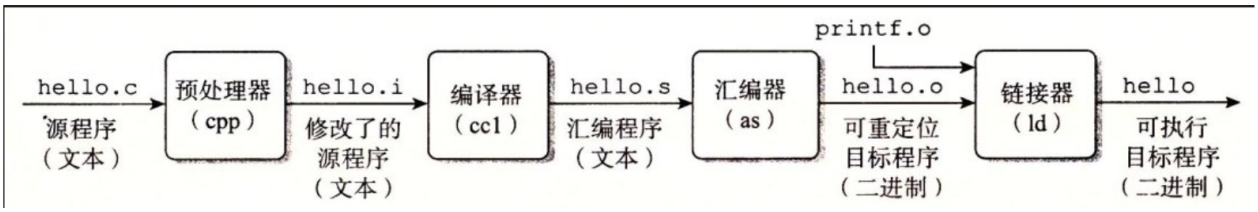
3.友元关系不具有传递性

113.动态编译和静态编译

就是动态链接和静态链接的区别

114.hello.c的编译过程

```
gcc -o hello hello.c
```



- 预处理阶段：处理以 # 开头的预处理命令；
- 编译阶段：翻译成汇编文件；
- 汇编阶段：将汇编文件翻译成可重定位目标文件；
- 链接阶段：将可重定位目标文件和 printf.o 等单独预编译好的目标文件进行合并，得到最终的可执行目标文件。

115.介绍一下几种典型的锁（还得多看）

读写锁

多个读者同时读；

写者必须互斥（只有一个写者写，读写不能同时进行）；

写者优先于读者（只要有写者，则后续读者就要等待）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁在加锁操作时涉及上下文的切换。互斥锁实际的效率还是可以让人接受的，加锁的时间大概100ns左右，而实际上互斥锁的一种可能的实现是先自旋一段时间，当自旋的时间超过阀值之后再将线程投入睡眠中，因此在并发运算中使用互斥锁（每次占用锁的时间很短）的效果可能不亚于使用自旋锁

条件变量

互斥锁一个明显的缺点是他只有两种状态：锁定和非锁定。总的来说互斥锁是线程间互斥的机制，条件变量则是同步机制

自旋锁

如果进线程无法取得锁，进线程不会立刻放弃CPU时间片，而是一直循环尝试获取锁，直到获取为止。如果别的线程长时期占有锁那么自旋就是在浪费CPU做无用功，但是自旋锁一般应用于加锁时间很短的场景，这个时候效率比较高

116.delete和delete[]

前者调用一次析构，后者调用数组中每个元素的析构

117.为什么不能把所有的函数写成内联函数

内联函数原地展开，比函数调用快，这才用它，如果过过于复杂，还不如调用

- 函数体内的代码比较长，将导致内存消耗代价

- 函数体内有循环，函数执行时间要比函数调用开销大

118.C++为什么没有垃圾回收机制？和JAVA不一样

- 首先，实现一个垃圾回收器会带来额外的空间和时间开销。你需要开辟一定的空间保存指针的引用计数和对他们进行标记mark。然后需要单独开辟一个线程在空闲的时候进行free操作。
- 垃圾回收会使得C++不适合进行很多底层的操作。

二、内存管理和新标准

1.类的对象存储空间

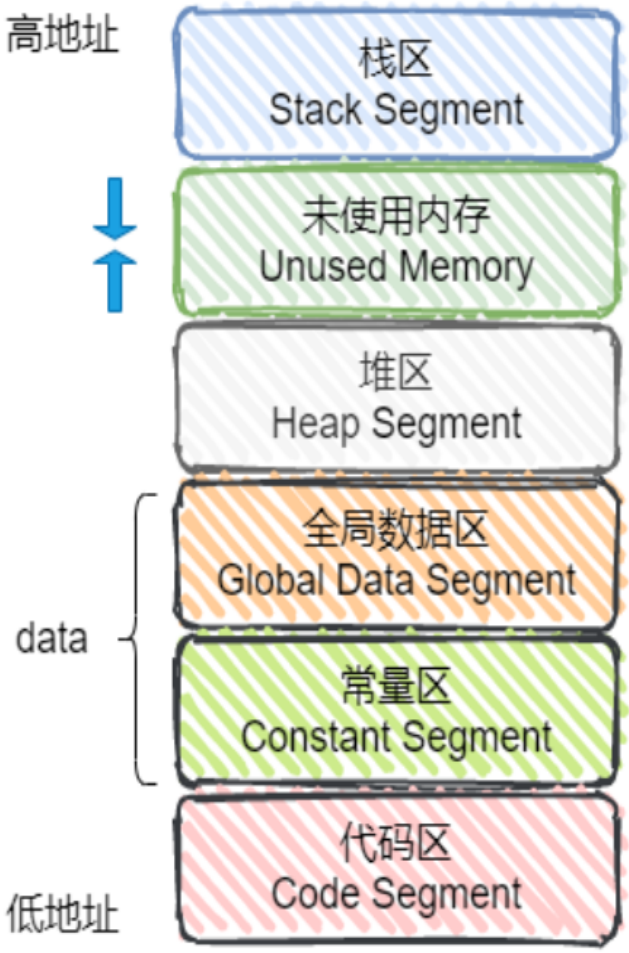
非静态成员的数据类型之和

编译器加入的额外成员变量（指向虚函数表的指针）

边缘对其优化的padding

空类（无非静态数据成员）的对象的size=1，作为基类时=0

2.简要说明C++的内存分区



全局/静态存储区(.bss .data)：全局变量和静态变量（局部和全局的static）被分配到同一块内存中，在以前的C语言中，全局变量和静态变量又分为初始化的和未初始化的，在C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如int型变量自动初始为0

常量存储区(.data)：这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

3.什么是内存池，如何实现

Memory pool，new和malloc申请的大小不定，频繁使用会产生碎片。可以预先申请备用，有需求时分出一部分，不够了再申请。

allocate 包装 malloc，deallocate包装free

一般是一次20*2个的申请，用一半留一半。

假设40个32b的区块，20个给程序用，19个维护，1个交出。

客户端有需求了，申请20*64b的空间，现在只有（19+1=20）个32b的，先将20个中的10*64b返回，1个交出，9个维护，内存池空了。

如果再有需求，就继续申请，内存池有用上，挂在0-15号的某一条链表上，否则重新申请。

4.你了解的C++内存管理？

5.C++中类的数据成员和成员函数内存分布情况

class由C的结构体发展而来，C语言的结构体只有成员变量。分配机制是相同的。

对象的大小和对象中数据成员的大小是一致的。成员函数不占用对象的内存，所有函数都存放在代码区，不管全局还是局部。

静态成员函数与一般成员函数的唯一区别就是没有this指针，因此不能访问非静态数据成员。静态函数也在代码区而不是全局区。

6.关于this指针知道什么？

this是类的指针，指向对象的首地址；

只在成员函数中使用，全局函数和静态成员函数中不能用；

在成员函数中才有定义，存储位置会因编译器的不同而不同。

用处：不是对象本身的一部分，不会影响sizeof的结果，编译器本身会加上this，是非静态成员函数的隐含形参。

使用：非静态成员函数中返回类对象本身，return *this

区分形参 this->n = n;

类的this指针有以下特点：

1.在构造成员函数时，默认定义一个类本身指针的this，在成员函数开始前构造，在成员函数结束后清除。2.效率高。

7.几个this的易混问题

1.创建时间

成员还是的开始和结束。

class和struct中没有方法==C的struct

typedef xxx 在栈里分配内存，this指针的值就是这块内存

new创建对象，放在eax中在堆里分配内存，内存块地址传给ecx

2.this指针放在哪里？

因编译器的不同有不同的位置。可能是栈，寄存器，全局变量。

在汇编中，一个值有三种出现形式：立即数，寄存器和内存变量值。因此不是放在寄存器就是放在内存中。

3.this指针如何传递类中的指针？绑定么，还是函数参数的首参数就是this指针？this指针如何找到“类实例化后的函数的”？

大多数编译器通过ecx计数寄存器传递this指针，这是潜规则。不同编译器要遵从一致的传参规则，否则obj无法匹配。

调用前，会将对象的地址放在eax中，this通过函数的参数的首参来传递。函数调用前this已经生成，“类实例后的函数”，没有这个说法。子啊实例化时，只分配类中的变量空间，不给函数分配空间，只要类一定义，函数就在那里。

4.this指针是如何访问类中的变量？

类似于结构的指针。

5.我们能直接使用一个对象this指针？

获得一个对象后，也不能通过对象使用this指针，在成员函数中才能使用。

6.每个类编译后，是否创建一个类中函数表保存函数指针，以便调用函数？

普通的类函数都不会创建的，只有虚函数才会。但即便是虚函数，如果知道调用哪一个，就能直接调用该函数，不会通过函数表中的指针。由于this的存在，只想了不同的对象，确保不同对象之间调用相同的函数可以互不干扰。

8.内存泄漏的后果？监测？解决方法？

1.由于疏忽错误造成程序无法释放掉已申请的内存空间，并非在物理上消失，而是应用程序分配某段内存后，由于设计的错误，失去了对该段内存的控制。

2.后果，小的内存泄漏可能不被注意，大的内存泄漏会使得性能下降至内存逐渐用完，导致另一个程序失败。

3.排除方法：Boundschecker，在debug版中运行CRT(C run-time libraries)，综合分析。

4.解决方法：智能指针

5.检查定位内存泄漏

加上头文件`#include <crtdbg.h>`

检查：main函数的最后一行`_CrtDumpMemoryLeaks()`,得到的是定位值和泄漏大小`{453}normal block at 0x02432CA8,868 bytes long`。

定位代码：main的第一行加上

`_CrtSetBreakAlloc(453)`，在453终端，调试程序，中断后查看调用堆栈。

9.在成员函数中调用delete this会出现什么问题？对象还能使用么？

类对象的内存中，代码不在那里，所以汉能继续执行代码。到那时会释放类的内存空间，之后只要不涉及this指针的内容，就能正常运行，一旦涉及到，如操作数据成员，调用虚函数，就会出现不可预期的问题。

10.为什么是不可预期的问题？

可能还没有回收到系统中去，所以此时这段内存是可以访问的，数据会很随机，访问虚函数表，指针无效的可能性非常高。

11.在类的析构中调用delete this，会发生什么？

delete就是调用析构，在析构中调用析构，无限递归，造成堆栈溢出，系统崩溃。

12.空类的大小是多少？

C++的空类不为0，vs是1哥字节。

C++标准指出，不允许一个对象的大小为0，不同的对象要有不同的地址。

带有虚函数的C++类的大小不为1，每一个对象有一个vptr只想虚函数表，具体大小根据指针大小决定。32位系统4字节，64位系统8字节。

13.说下几种情况下的类的大小是多少？

当该空白类作为基类时，该类的大小就优化为0了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化

有指针的话，就是指针的大小。

```
class A { virtual Fun(){} };
int main(){
  cout<<sizeof(A)<<endl;// 输出 4(32位机器)/8(64位机器);
  A a;
  cout<<sizeof(a)<<endl;// 输出 4(32位机器)/8(64位机器);
  return 0;
}
```

静态成员和普通函数不占用类的大小。

14.this指针调用成员变量时，堆栈会发生什么变化？

this指针作为隐含参数自动传给函数，this指针首先入栈，后面参数从右向左入栈，最后函数返回地址入栈。

15.类对象的大小受哪些因素影响？

成员函数和静态成员不占据类的大小，内存对齐会另外分配空间，虚函数会插入vptr指针，该类是基类的话，子类也会相应的扩展。

C++11新标准

16.新特性有哪些

- 类和结构体的中初始化列表
- Lambda 表达式（匿名函数）
- std::forward_list（单向链表）
- 右值引用和move语义
- nullptr代替null；
- 引入了auto和decltype实现类型推导
- 基于范围的 for 循环for(auto& i : res){}

17.auto decltype 和decltype(auto)

自动类型推断，auto必须有初始化的值

decltype用表达式推出类型值，但是不计算表达式

decltype(auto)是C++14新增的类型指示符，可以用来声明变量以及指示函数返回类型。在使用时，会将“=”号右边的表达式替换掉auto，再根据decltype的语法规则来确定类型

18.NULL和nullptr的区别

C中 NULL=（void*) 0 C++中NULL=0

C	NULL=(VOID*) 0
C++	NULL=0

C++中有重载，那么NULL和0就不分了，所以要引入nullptr作为指针。但在使用的时候，还是要指明是哪个类型的nullptr

```
void fun(char* p)
{
    cout<< "char* p" <<endl;
}
void fun(int* p)
{
    cout<< "int* p" <<endl;
}
void fun(int p)
{
    cout<< "int p" <<endl;
}
int main()
{
    fun((char*)nullptr);//语句1
    fun(nullptr);//语句2
    fun(NULL);//语句3
    return 0;
}
//运行结果:
//语句1: char* p
//语句2:报错，有多个匹配
//3: int p
```

19.智能指针的原理，常见的智能指针及实现

SHARED_PTR	**采用计数器，多个智能指针指向同一个对象，多一个指针，引用计数+1，少一个就-1，计数为0就自动释放动态分配的资源。（赋值的时候，左边的-1，右边的+1，拷贝的时候指针数一起拷贝过去+增加的计数）**
unique_ptr	一个非空的unique_ptr总是拥有它所指向的资源，独占，不支持拷贝和复制，不能用在STL标准容器中，局部变量返回值可以因为马上会销毁。如果拷贝，结束后释放两次，程序崩溃。
weak_ptr	弱引用。用来打破环形引用。配合shared_ptr，只引用不计数。s析构之后，不管weak_ptr是否引用内存，内存都会被释放，不保证指向的内存是有效的，使用前用lock()检查是否为空指针。它只可以从一个 shared_ptr 或另一个 weak_ptr 对象构造, 它的构造和析构不会引起引用记数的增加或减少
auto_ptr	解决有异常抛出时发生内存泄漏的问题。因为发生异常而无法正常释放内存。有拷贝语义，拷贝后源对象无效，会有严重的问题，不能进行赋值。u无拷贝但有移动语义，std::move可以传递

20.lambda函数

1.编写内嵌的匿名函数，替换独立函数或者函数对象，是一个可调用的对象

2.格式，默认返回的是void，如果有除了return的语句就要写返回类型

```
\[&/=, para](int x, int  y) mutbale ->int {if()
return xx}
[参数列表，可引用可值传递](需要的参数)->返回值类型{语句}
```

或者直接定义

```
auto f = [](){}
还有可变lambda，可以改变捕获的值
auto f =[]() mutable{}
```

如果调用频繁，写成函数

如果捕获参数太多，用bind，在functional里，多个参数转化成一元谓词。

21.智能指针的作用

1.管理堆内存，防止内存泄漏或者二次释放

2.在中，shared_ptr的引用计数是线程安全的，到那时对象的读取需要加锁。

3.初始化。是个模板类

```
shared_ptr<int> p4(new int(4));
不能隐式转换 是explicit的
shared_ptr<int> p4 = new int(4);是错误的
shared_ptr<int> p1 = make_shared<int>(9)
```

22.了解的auto_ptr

auto_ptr构造的时候取得某个对象的控制权，析构时释放，自身析构时会释放指向空间的内存。所以不会有内存泄漏。

构造函数是explicit的，阻止一般指针转向本身的隐式转换，但是支持所拥有指针类型之间的隐式转换。

避免多个auto_ptr对象管理同一个指针。

删除时delete，不能管理数组。

T* get() 获得auto_ptr所拥有的指针，T* release()释放所有权。

23.智能指针的循环引用

在实际编程过程中，应该尽量避免出现智能指针之前相互指向的情况，如果不可避免，可以使用使用弱指针——weak_ptr，它不增加引用计数，只要出了作用域就会自动析构。

24.手写实现智能指针类需要实现哪些函数？

1.一个构造，拷贝构造，赋值构造，析构，移动函数

2.是一个数据类型，用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。设置计数器需要一个构造函数，新增对象需要一个构造函数，析构函数减少计数和释放内存，覆写赋值运算符，新旧指针之间拷贝。

25.循环引用怎么办？

两个对象互相使用一个shared_ptr成员变量指向对方。弱引用能检测到管理的对象是否被释放，避免访问非法内存。

三、STL模板库和其他问题

STL模板库

1.STL？

广义分为3类：算法，容器和迭代器。

容器：关联式和序列式容器，迭代器就是不暴露容器内部结构的情况下对容器的遍历。

2.trivial destructor

“trivial destructor”一般是指用户没有自定义析构函数，而由系统生成的，这种析构函数在《STL源码解析》中成为“无关痛痒”的析构函数。

首先利用value_type()获取所指对象的型别，再利用**type_traits判断该型别的析构函数是否trivial，若是(**true_type)，则什么也不做，若为(__false_type)，则去调用destory()函数。

3.使用智能指针管理内存资源，RAII是怎么回事？

1.RAII全称是“Resource Acquisition is Initialization”，直译过来是“资源获取即初始化”，也就是说在构造函数中申请分配资源，在析构函数中释放资源。将资源和对象的生命周期绑定。

2.智能指针是RAII最具代表的实现。可以实现自动的内存管理，不需要担心内存泄漏。

4.迭代器：++it和it++哪个好？

前置返回一个引用，后置返回一个对象，后置效率低。

```
// ++i实现代码为：
int& operator++()
{
  *this += 1;
  return *this;
}

//i++实现代码为：
int operator++(int)
{
   int temp = *this;
   ++*this;
   return temp;
}
```

5.说一下C++左值引用和右值引用

右值：无法获取地址的对象，常量，函数返回值，lambda表达式等，无法获取地址！=不可改变，右值引用就可以更改右值，得到的是右值临时对象存放的地址。

右值引用：

- 右值引用让右值重获新生，生命周期延长；
- 独立于左值和右值，右值引用类型的变量可能是左值或右值
- T&&t 发生在自动类型推断的时候，初始化决定了它是左值还是右值

右值分类：纯右值（prvalue, pure Rvalue）和将亡值（xvalue, expiring value)

prvalue:指的是临时变量和不跟对象关联的字面量值

xvalue:11新增的，通常是将要被移动的对象，

比如返回右值引用T&&的函数返回值、std::move的返回值，或者转换为T&&的类型转换函数的返回值

“盗取”其他变量内存空间的方式获取到的值。在确保其他变量不再被使用、或即将被销毁时，通过“盗取”的方式可以避免内存空间的释放和分配，能够延长变量值的生命期。

左值：

左值引用是具名变量值的别名，而右值引用则是不具名（匿名）变量的别名

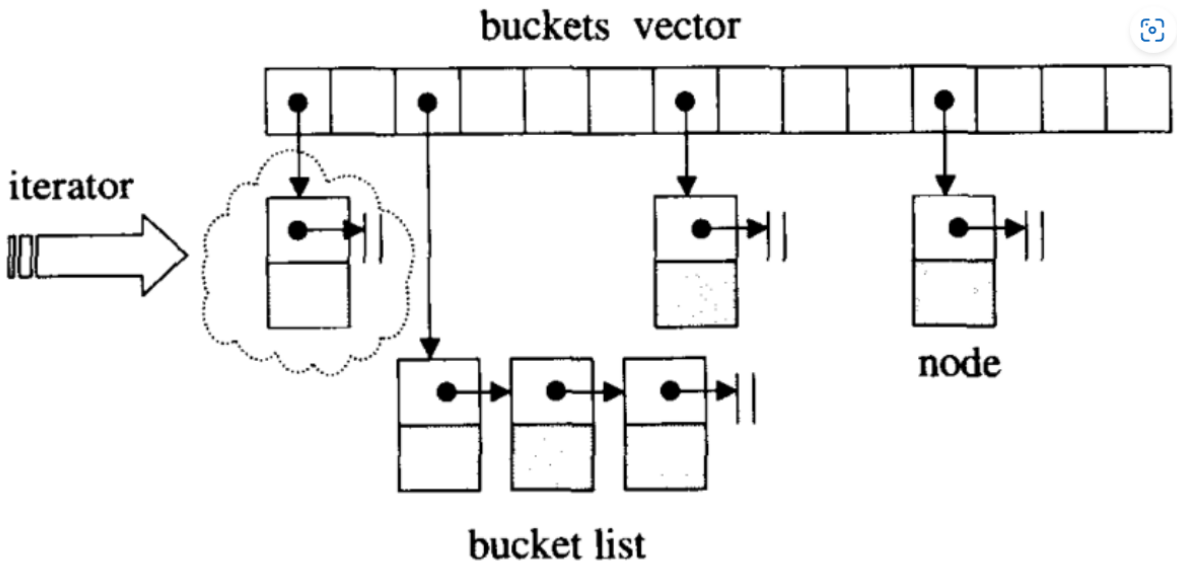
左值引用通常也不能绑定到右值，但**常量左值引用**是个“万能”的引用类型。它可以接受非常量左值、常量左值、右值对其进行初始化。不过常量左值所引用的右值在它的“余生”中只能是只读的。

```
#include <bits/stdc++.h>
using namespace std;

template<typename T>
void fun(T&& t)
{
    cout << t << endl;
}

int getInt()
{
    return 5;
}

int main() {
int a = 10;
int& b = a;  //b是左值引用
int& c = 10;  //错误，c是左值不能使用右值初始化
int&& d = 10;  //正确，右值引用用右值初始化
int&& e = a;  //错误，e是右值引用不能使用左值初始化
const int& f = a; //正确，左值常引用相当于是万能型，可以用左值或者右值初始化
const int& g = 10;//正确，左值常引用相当于是万能型，可以用左值或者右值初始化
const int&& h = 10; //正确，右值常引用
const int& aa = h;//正确
int& i = getInt();  //错误，i是左值引用不能使用临时变量（右值）初始化
int&& j = getInt();  //正确，函数返回值是右值
```

```
fun(10); //此时fun函数的参数t是右值
fun(a); //此时fun函数的参数t是左值
return 0;
}
```

6.STL中hashtable的实现

使用的是开链法解决hash冲突的问题。



自己定义的由hashtable_node数据结构组成的linked-list，而bucket聚合体本身使用vector进行存储。hashtable的迭代器只提供前进操作，不提供后退操作。

设计的bucket的数量上，内置了28个质数，创建的时候选择合适大小作为容量，也就是vector的长度，每个linked-list的长度也等于hashtable的容量。如果插入的元素个数超过了bucket的容量，就要重建table，找出下一个质数，再次创建和计算。

7.简单说一下traits技法

利用“内嵌型别”的编程技巧与编译器的template参数推导功能，增强C++未能提供的关于型别认证方面的能力。常用的有iterator_traits和type_traits

iterator_traits

被称为**特性萃取机**，能够方便的让外界获取以下5种型别：

- value_type：迭代器所指对象的型别
- difference_type：两个迭代器之间的距离
- pointer：迭代器所指向的型别
- reference：迭代器所引用的型别
- iterator_category：三两句说不清楚，建议看书

type_traits

关注的是型别的**特性**，例如这个型别是否具备non-trivial defalt ctor（默认构造函数）、non-trivial copy ctor（拷贝构造函数）、non-trivial assignment operator（赋值运算符）和non-trivial dtor（析构函数），如果答案是否定的，可以采取直接操作内存的方式提高效率。

8.STL的两级空间配置器

反复开辟和释放会产生外部碎片，用二级空间配置器开辟小块内存。

于是就设置了二级空间配置器，**当开辟内存<=128bytes时，即视为开辟小块内存，则调用二级空间配置器。**

关于STL中一级空间配置器和二级空间配置器的选择上，一般默认**选择的为二级空间配置器**。 如果大于128字节再转去一级配置器器。

一级空间配置器

重要的函数就是allocate、deallocate、reallocate 。 一级空间配置器是以malloc()，free()，realloc()等C函数执行实际的内存配置

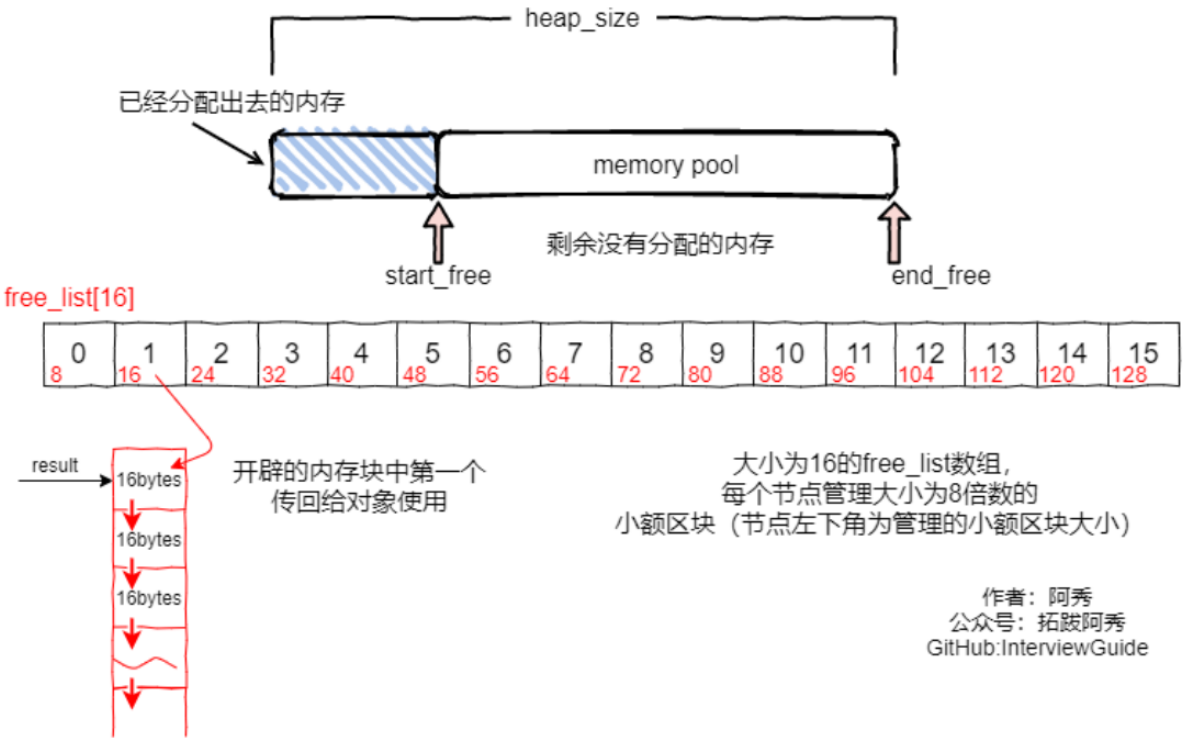
1.分配，失败的话调用处理函数

2.如果定义了处理函数，调用，失败的话返回

3.处理函数成功，就结束，继续分配。

二级空间配置器

1。维护16条链表，free_list[0-15]，最小为八字节，以8字节递增，最大是128字节，传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表。不为空的话拔出链表，将已经拔出的指针后移一位。挂在上面的都是可以用的。



2.对应的free_list为空，先看内存池是否为空，不过不为空

- 检查空间是否够20个节点，足够的话取出20个，1个分配给用户，19个当做自由链表中的区块挂在相应的free_list下，如果下次再有相同大小的内存需求时，可直接拔出。
- 不够20个，是否满足1个，有的话分配一个给用户，剩余的挂在free_list中
- 一个都不够，挂在list中，再去申请空间

3.内存池为空，申请20+20=40块的空间，一半用一般放在内存池中。

4.分配不成功，heap没空间了，去free_list中找比当前节点大的空间，如果有，就拔出来使用，否则就要去使用一级空间配置器了。

缺点：

1.内部碎片问题，用1字节申请8字节，7字节浪费了。

2.二级中也会有狭义的内存池，程序执行过程中不会释放

- 不断开辟小的内存会让整个heap空间都被挂在自由链表上，这时候开辟大块内存就会失败
- 如果自由链表上的内存没有被使用，当前进程占着程序不放，别的进程又申请不到空间，会引发多种问题。

9.vector和list的区别和应用？怎么找到倒数第二个元素？

vector类似数组，内存空间连续。可以动态增长，随机存取比较高效O(1)，但是插入和删除比较复杂，会产生大量的拷贝，时间复杂度O(n).

list双向链表，内存空间不连续，随机访问不方便，插入和删除很方便，一个节点包括值，前向指针和后向指针。

vector的迭代器使用过就失效了，list还能接着用？

10.STL中vector删除元素，迭代器怎么变化？为什么是两倍扩容？释放空间？

size() 有多少元素

capacity() 总共能存多少元素

两者相等时说明空间用尽，如果要再添加新元素就会扩容。

可以先预先给定一个内存大小

reserve() 不让变化的那么频繁，可能还是野的，用[]访问可能会越界

resize改变元素数量，不改变容器大小。

不同的编译器，扩容大小不同，vs下是1.5倍，gcc是2倍。

在析构的时候才会释放空间。申请了多少就是多少，元素变化不影响内存分配的原始值。

可以用deque实现空间动态缩小。

两个指令可以清除

```
vector(Vec).swap(Vec); //将Vec中多余内存清除；
vector().swap(Vec); //清空Vec的全部内存；
```

11.容器内删除一个元素

1.顺序容器（vector，deque等)

erase会使被删除的迭代器失效，还会使被删除元素之后的所偶迭代器失效（list除外），所以不能erase(it++)，但是erase的返回值是下一个有效迭代器

```
it = c.erase(it)
```

2.关联式容器(map, set,multimap,multiset)

使被删除的迭代器失效，但是返回值是void

```
c.erase(it++)
```

12.STL迭代器如何实现

迭代器是一种抽象的设计理念，通过它可以在不了解容器内部原理的情况下遍历容器，还能够作为容器和STL算法的粘合剂。

作用是接口，保存一个与容器相关的指针，重载各种运算符，类似于智能指针。

五种迭代器：value type、difference type、pointer、reference、iterator catagoly;

13.map，set的实现？红黑树怎么能够同时实现这两种容器？为什么用红黑树？

他们的底层都是以红黑树的结构实现，因此插入删除等操作都在O(logn时间内完成，因此可以完成高效的插入删除；

在这里我们定义了一个模版参数，如果它是key那么它就是set，如果它是map，那么它就是map；底层是红黑树，实现map的红黑树的节点数据类型是key+value，而实现set的节点数据类型是value

因为map和set要求是自动排序的，红黑树能够实现这一功能，而且时间复杂度比较低。

14.如何在共享内存上使用STL标准库？

想像一下把STL容器，例如map, vector, list等等，放入共享内存中，IPC一旦有了这些强大的通用数据结构做辅助，无疑进程间通信的能力一下子强大了很多。不用再为共享内存设计其他的数据结构了

进程A在共享内存中放入了数个容器，进程B怎么找到呢？

固定地址或者map映射。

15.map插入方式有几种？

```
mapStudent.insert(pair<int, string>(1, "student_one"));

mapStudent.insert(map<int, string>::value_type (1, "student_one"));

mapStudent.insert(make_pair(1, "student_one"));

mapStudent[1] = "student_one";
```

16. unordered_map(hash_map)和map的区别，hash_map如何解决冲突以及扩容？

UNORDERED_MAP	MAP
两者都是key-value键值对，到那时不会根据key的大小进行排序	
内部的元素是无序的	有序的，按照二叉树搜索树进行存储
map的key需要定义operator<	要定义hash_value并且重载operator==
如果是自定义类型的话，就要自己定义上面两个了	
底层是hash_table，开链法实现	底层是二叉树

什么时候扩容：当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

17.vector越界访问下标，map越界访问下标？vector删除元素时会不会释放空间？

vector做边界检查，具体实现看IDE

map使用key作为下标，返回相应的值，不存在的话就插入

erase()，只能删除内容，不能改变容量大小。

erase成员函数删除了迭代器指向的元素，相当于一个智能指针

clear清空内容，不能改变容量，如果在删除的时候同时释放内存，可以选择deque。

18.map中的[]和find的区别

map的下标运算符[]的作用是：将关键码作为下标去执行查找，并返回对应的值；如果不存在这个关键码，就将一个具有该关键码和值类型的默认值的项插入这个map。

map的find函数：用关键码执行查找，找到了返回该位置的迭代器；如果不存在这个关键码，就返回尾迭代器。

19.STL中list和deque的区别

list插入操作原有的list迭代器失效；

deque双向插入，可以随着元素的变化而申请和释放内存。

queue不提供迭代器

20.STL中的allocator、deallocator

1.以及配置器直接使用malloc，free和relloc，第二级视情况采用不同的策略。

2.二级配置器主动调整内存需求至8的倍数，并维护16个free-list，各自管理一个区块；

3.如果free-list中有可用区块，就使用，否则refill重新给free-list分配空间

4.空间释放函数deallocate()，如果大于128，直接调用一级配置器，小于128就找到队形的free-list释放内存

21.STL中hash_table扩容发生什么？

6.STL中hashtable的实现

1.hash_table中的元素称为桶，由桶所连接的元素称为节点，存入桶元素的容器是vector，vector有动态扩容能力。

2.向前操作：当前节点出发，前进一个节点，节点在list内，用next就能完成，如果list是尾端，就跳至下一个bucket身上，正是指向下一个list的头部节点。

22.常见容器性质总结

VECTOR	数组，快速随机访问
list	双向链表，快速增删
deque	一个中控器+多个缓冲区，首尾快速增删，随机访问，有很多堆，堆与堆之间用指针指向，堆内部保存好几个元素，相当于list+vector
stack/queue	封闭头部/尾部
priority_queue	底层是vector容器，用堆位处理规则管理容器实现
set/map	红黑树，有序不重复
multiset/multimap	红黑树，有序可重复
unordered_xxx	hash表，全部无序，multi可重复

23.vector的增加删除都是怎么做的？为什么是1.5或者是2倍？

对比可以发现采用采用成倍方式扩容，可以保证常数的时间复杂度，而增加指定大小的容量只能达到O(n)的时间复杂度，因此，使用成倍的方式扩容。

pop_back和erase改变大小，remove一般不会改变大小。

24.STL每种容器对应的迭代器

容器	迭代器
vector、deque	随机访问迭代器
stack、queue、priority_queue	无
list、(multi)set/map	双向迭代器
unordered_(multi)set/map、forward_list	前向迭代器

25.STL迭代器失效的情况？

vector

插入：

尾后插入，尾迭代器失效，size==capacity时，所有迭代器失效

中间插入：插入点之后的迭代器全失效

删除：

尾后删除，尾迭代失效

中间删除，删除位置之后的失效

deque类似

list删除之后，删除节点失效，返回下一个有效的迭代器

map/set是红黑树，不影响其他迭代器，递增方法获得下一个迭代器map.erase(it++);

unordered_的迭代器意义不大，rehash之后全部失效

26.vector实现

vector是一块连续的线性空间，重新分配空间，移动数据，释放原空间。Vector扩容倍数与平台有关，在Win + VS 下是 1.5倍，在 Linux + GCC 下是 2 倍

27.slist实现

单向链表，只有单向的forward iterator，耗用的空间小，操作更快。

只有`insert_after()`和`erase_after`,只提供`push_front()`

需要注意的是C++标准委员会没有采用slist的名称，forward_list在C++ 11中出现，它与slist的区别是没有size()方法。

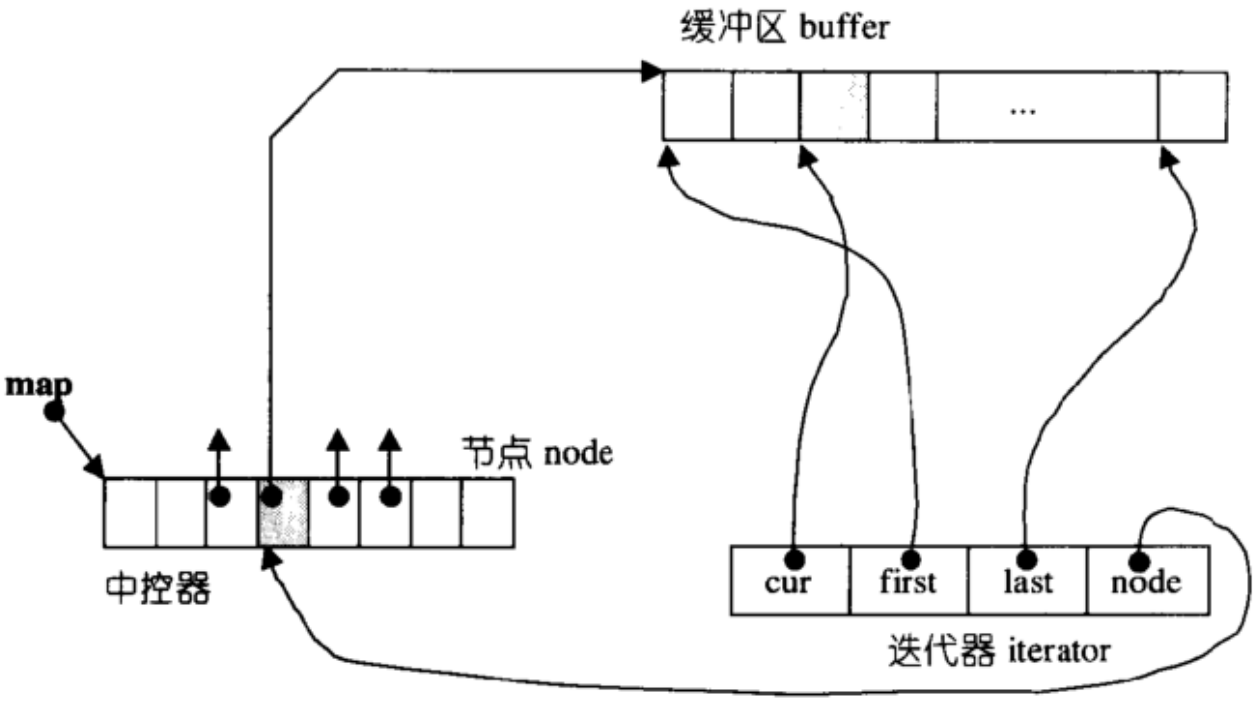
28.list实现

双向，是环形链表，一个指针就能完整表现整个链表。

此外list也是一个环形链表，因此只要一个指针便能完整表现整个链表。list中node节点指针始终指向尾端的一个空白节点，因此是一种“前闭后开”的区间结构

29.deque的实现

没有容量的概念，排序的话可以先复制到vector排完以后再复制回去。



如果使用完毕node和node指向的缓存区，那么可以分配新的map，容量是2*当前容量+2，删除元素时，进行析构。

map->node->buffer

iterator->buffer

iterator->node

30.STL中stack和queue的实现

默认是由deque实现

queue可以作为list的底层容器

31.Heap的实现

heap（堆）并不是STL的容器组件，是priority queue（优先队列）的底层实现机制，因为binary max heap（大根堆）总是最大值位于堆的根部，优先级最高。

binary heap本质是一种complete binary tree（完全二叉树），整棵binary tree除了最底层的叶节点之外，都是填满的，但是叶节点从左到右不会出现空隙

完全二叉树内没有任何节点漏洞，是非常紧凑的，这样的一个好处是可以使用array来存储所有的节点，因为当其中某个节点位于i处，其左节点必定位于$2i$处，右节点位于$2i+1$处，父节点位于$i/2$（向下取整）处。这种以array表示tree的方式称为隐式表述法。

插入元素相当于构造了一个大根堆，先把元素放在叶子节点，再去调整进行上溯。

弹出元素是将根节点放到vector的最末尾处，然后调整，下溯。

排序就是不断pop，将大顶堆的根节点放在最后面，不断重复此过程。

32.priority_queue实现

优先队列，像队列一样入队出队，但是出队顺序是根据权值决定的，权值大的先出。

默认情况下，priority_queue使用一个max-heap完成，底层容器使用的是一般为vector为底层容器，堆heap为处理规则来管理底层容器实现 。priority_queue的这种实现机制导致其不被归为容器，而是一种容器配接器。

33.set的实现

键值不重复，自动生序排列。红黑树是底层机制

- 每个节点不是红色就是黑色
- 根结点为黑色
- 如果节点为红色，其子节点必为黑
- 任一节点至（NULL）树尾端的任何路径，所含的黑节点数量必相同

34.map的实现

RB-tree红黑树

insert_unique() 而不是insert_equal() 是multimap使用的

- 下标操作可以是左值（修改内容）也可以是右值（获取实值）
- 运行过程，先根据键值和实值构造一个临时对象，再将对象插入map中去，返回一个pair，如果插入成功返回true，如果失败返回false，取到实值

35.set/map，multimap/multiset区别

set的key和value是一样的，保存的也是两份元素

multimap和map的唯一区别就是：multimap调用的是红黑树的insert_equal(),可以重复插入而map调用的则是独一无二的插入insert_unique()，multiset和set也一样，底层实现都是一样的，只是在插入的时候调用的方法不一样。

36.红黑树概念

1、它是二叉排序树（继承二叉排序树特性）：

- 若左子树不空，则左子树上所有结点的值均小于或等于它的根结点的值。
- 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值。
    - 左、右子树也分别为二叉排序树。

2、它满足如下几点要求：

- 树中所有节点非红即黑。
- 根节点必为黑节点。
- 红节点的子节点必为黑（黑节点子节点可为黑）。
- 从根到NULL的任何路径上黑结点数相同。

3、查找时间一定可以控制在O(logn)。

37.unordered_map和map的区别和应用场景

MAP(有序)	UNORDERED_MAP(无序)
红黑树，红黑树的查询和维护时间复杂度均为$O(logn)$，但是空间占用比较大，因为每个节点要保持父节点、孩子节点及颜色的信息	是C++ 11新添加的容器，底层机制是哈希表，通过hash函数计算元素位置，其查询时间复杂度为O(1)，维护时间与bucket桶所维护的list长度有关，但是建立hash表耗时较大

38.hash_table中解决冲突的方法

记住前三个：

线性探测

使用hash函数计算出的位置如果已经有元素占用了，则向后依次寻找，找到表尾则回到表头，直到找到一个空位

开链

每个表格维护一个list，如果hash函数计算出的格子相同，则按顺序存在这个list中

再散列

发生冲突时使用另一种hash函数再计算一个地址，直到不冲突

二次探测

使用hash函数计算出的位置如果已经有元素占用了，按照1^2、2^2、3^2...的步长依次寻找，如果步长是随机数序列，则称之为伪随机探测

公共溢出区

一旦hash函数计算的结果相同，就放入公共溢出区

其余问题

1.C++的多态如何实现

加上virtual关键字，在子类中重写该函数。运行时调用父类的就是父类的函数，调用的是子类对象就是子类的函数。

底层原理是虚表和虚基表指针。

虚表：虚函数表**vtable**的缩写，类中含有virtual关键字修饰的方法时，编译器会自动生成虚表。**定义类的时候产生**

虚表指针：在含有虚函数的类实例化对象时，对象地址的前四个字节存储的指向虚表的指针。**实例化对象的时候产生**

（1） 编译器在发现基类中有虚函数时，会自动为每个含有虚函数的类生成一份虚表，该表是一个一维数组，虚表里保存了虚函数的入口地址

（2）编译器会在每个对象的前四个字节中保存一个虚表指针，即**vptr**，指向对象所属类的虚表。在构造时，根据对象的类型去初始化虚指针vptr，从而让vptr指向正确的虚表，从而在调用虚函数时，能找到正确的函数

（3）所谓的合适时机，在派生类定义对象时，程序运行会自动调用构造函数，在构造函数中创建虚表并对虚表初始化。在构造子类对象时，会先调用父类的构造函数，此时，编译器只“看到了”父类，并为父类对象初始化虚表指针，令它指向父类的虚表；当调用子类的构造函数时，为子类对象初始化虚表指针，令它指向子类的虚表

（4）当派生类对基类的虚函数没有重写时，派生类的虚表指针指向的是基类的虚表；当派生类对基类的虚函数重写时，派生类的虚表指针指向的是自身的虚表；当派生类中有自己的虚函数时，在自己的虚表中将此虚函数地址添加在后面

2.为什么析构函数一般写成虚函数

如果析构函数不被声明成虚函数，则编译器实施静态绑定，在删除基类指针时，只会调用基类的析构函数而不调用派生类析构函数，这样就会造成派生类对象析构不完全，造成内存泄漏。

3、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

构造函数：

- 虚函数对应一个vtable(虚函数表)，类中存储一个vptr指向这个vtable。如果构造函数是虚函数，就需要通过vtable调用，可是对象没有初始化就没有vptr，无法找到vtable，所以构造函数不能是虚函数。

析构函数：

- 析构函数可以为虚函数，并且一般情况下基类析构函数要定义为虚函数。
- 只有在基类析构函数定义为虚函数时，调用操作符delete销毁指向对象的基类指针时，才能准确调用派生类的析构函数（从该级向上按序调用虚函数），才能准确销毁数据。
- **析构函数可以是纯虚函数**，含有纯虚函数的类是抽象类，此时不能被实例化。但派生类中可以根据自身需求重新改写基类中的纯虚函数。

4.基类的虚函数表放在内存的什么区？虚函数表vptr的初始化时间？

C++中虚函数表位于只读数据段（.rodata），也就是C++内存模型中的常量区；而虚函数则位于代码段（.text），也就是C++内存模型中的代码区。

虚函数表是全局共享的，编译时完成构造，不是函数，只是虚函数的地址。数量确定，不需要动态生成。

5.模板函数和模板类的特例化

引入原因

编写单一的模板，它能适应多种类型的需求，使每种类型都具有相同的功能，但对于某种特定类型，如果要实现其特有的功能，单一模板就无法做到，这时就需要模板特例化

定义

对单一模板提供的一个特殊实例，它将一个或多个模板参数绑定到特定的类型或值上

（1）模板函数特例化

必须为原函数模板的每个模板参数都提供实参，且使用关键字template后跟一个空尖括号对<>，表明将原模板的所有模板参数提供实参。

本质是实例化一个模板，而不是重载。特例化放在后面，模板放在前面。

（2）类模板特例化

原理类似函数模板，**不过在类中，我们可以对模板进行特例化，也可以对类进行部分特例化。**对类进行特例化时，仍然用template<>表示是一个特例化版本

6.构造函数、析构函数、虚函数可否声明为内联函数?

《Effective C++》中所阐述的是：**将构造函数和析构函数声明为inline是没有什么意义的，即编译器并不真正对声明为inline的构造和析构函数进行内联操作，因为编译器会在构造和析构函数中添加额外的操作（申请/释放内存，构造/析构对象等），致使构造函数/析构函数并不像看上去的那么精简。**

class中的函数是默认inline的，编译器也只是选择性的inline。

将虚函数声明为inline，要分情况讨论

当是指向派生类的指针（多态性）调用声明为inline的虚函数时，不会内联展开；当是对象本身调用虚函数时，会内联展开，当然前提依然是函数并不复杂的情况下。

7.C++模板是什么，底层怎么实现？

1.编译器并不是把函数模板处理成能够处理任意类的函数；编译器从函数模板通过具体类型产生不同的函数；编译器会对函数模板进行两次编译：在声明的地方对模板代码本身进行编译，在调用的地方对参数替换后的代码进行编译。

2.这是因为函数模板要被实例化后才能成为真正的函数，在使用函数模板的源文件中包含函数模板的头文件，如果该头文件中只有声明，没有定义，那编译器无法实例化该模板，最终导致链接错误。

8.构造函数为什么不能是虚函数？析构函数为什么要虚函数？

1.存储角度：虚函数要有虚函数表，还要有能指向它的vptr，但是对象实例化之后才有vptr，在构造的时候，要通过vptr找vtable得到构造函数，但是现在还没有实例化，就无法开始这个过程。

2.使用角度：虚函数就是要再信息不全的情况下，重载函数得到相应的调用，构造函数本身就要初始化实例，要实不要虚。虚函数为了调用子类的函数，但是构造本身就是要自己创建的。

C++中的基类采用virtual虚析构函数是为了防止内存泄漏。

具体地说，如果派生类中申请了内存空间，并在其析构函数中对这些内存空间进行释放。假设基类中采用的是非虚析构函数，当删除基类指针指向的派生类对象时就不会触发动态绑定，因而只会调用基类的析构函数，而不会调用派生类的析构函数。那么在这种情况下，派生类中申请的空间就得不到释放从而产生内存泄漏。

最好不要把许析构函数定义为纯虚析构函数。

9.析构函数的作用？怎么起作用？

1.构造函数：默认调用，传递实参

2.析构：撤销对象的一些特殊任务处理，可以是释放对象分配的内存空间，五参数，无返回值，不能重载，只有一个，自动调用。

10.构造函数和析构函数中可以调用虚函数么？

不提倡，语法上没问题，但是达不到目的

1.构造：父类在子类之前构造，子类还未初始化，此时如果调用子类的虚函数不安全

2.析构：子类在父类之前析构，调用父类的析构时，子类已经销毁，调用子类的虚函数无意义。

11.构造析构的执行顺序？构造函数和拷贝构造的内部都干了啥？

构造函数顺序：基类，成员类，派生类

析构：派生类，成员类，基类。

12.构造函数析构函数可否抛出异常

1.析构已经完成的对象，如果构造没有完成，发生异常，那控制权转出构造函数之外，会造成内存泄漏

2.auto_ptr取代指针类成员，强化构造函数，免除了抛出异常时发生资源泄露的危机，不再需要在析构函数中手动释放资源

3.控制权因为异常离开析构函数，此时有另一个异常处于作用状态，C++会调用terminate函数结束程序

4.析构时异常，析构函数执行不全。

13.构造函数一般不定义为虚函数的原因

构造函数的特点，虚函数的特点，两者的矛盾点。

14.类在什么时候析构？

1.对象生命周期结束，或者倍被销毁时

2.delete指向对象的指针时，或delete指向对象的基类类型指针，二期基类析构函数是虚函数时

3.对象i是对象o的成员，o的析构函数被调用时，对象i的析构函数也被调用。

15.构造函数的几种关键字

default:关键字可以显式要求编译器生成合成构造函数，防止在调用时相关构造函数类型没有定义而报错

delete:关键字可以删除构造函数、赋值运算符函数等，这样在使用的时候会得到友善的提示

0：将虚函数定义为纯虚函数（纯虚函数无需定义，= 0只能出现在类内部虚函数的声明语句处；当然，也可以为纯虚函数提供定义，函数体可以定义在类的外部也可以定义在内部。

16.构造函数、拷贝构造函数和赋值操作符的区别

构造函数

对象不存在，没用别的对象初始化，在创建一个新的对象时调用构造函数

拷贝构造函数

对象不存在，但是使用别的已经存在的对象来进行初始化

赋值运算符

对象存在，用别的对象给它赋值，这属于重载“=”号运算符的范畴，“=”号两侧的对象都是已存在的

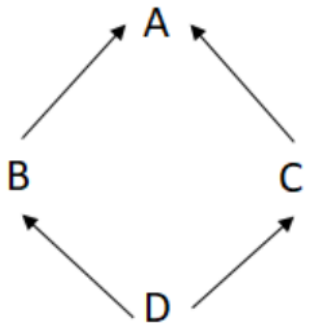
17.拷贝构造和赋值运算符重载的区别？

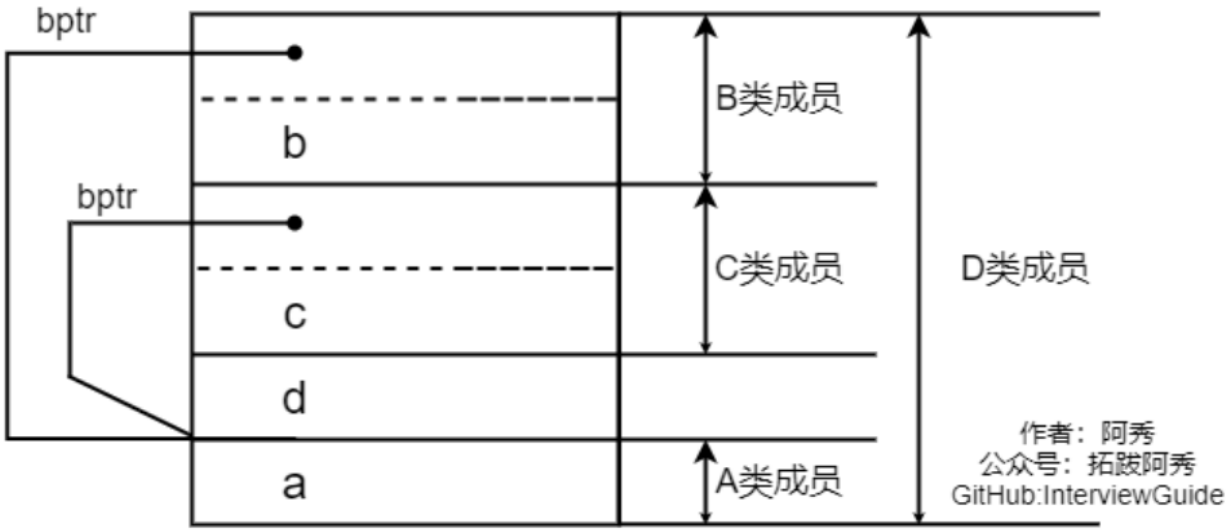
拷贝	赋值
函数	运算符重载
生成新的类对象	不生成
构造新的类对象，不检查两者是否相同	要检查，原来对象有内存就要释放掉
形参传递	并不是所有出现=的地方都是使用赋值运算符

```
Student s;
Student s1 = s;     // 调用拷贝构造函数
Student s2;
s2 = s;     // 赋值运算符操作
```

18.虚拟继承

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

class A{}
class B : virtual public A{};
class C : virtual public A{};
class D : public B, public C{};

int main()
{
    cout << "sizeof(A)：" << sizeof A <<endl; // 1，空对象，只有一个占位
    cout << "sizeof(B)：" << sizeof B <<endl; // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(C)：" << sizeof C <<endl; // 4，一个bptr指针，省去占位,不需要对齐
    cout << "sizeof(D)：" << sizeof D <<endl; // 8，两个bptr，省去占位,不需要对齐
}

```

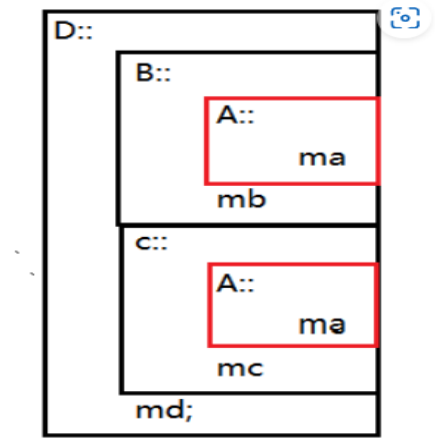




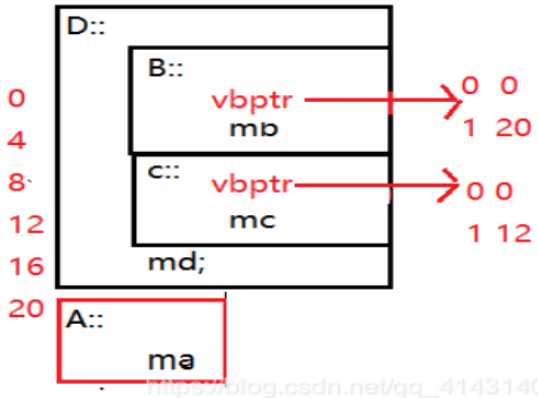
虚拟继承的情况下，无论基类被继承多少次，只会存在一个实体虚拟继承基类的子类中，子类会增加某种形式的指针，或者指向虚基类子对象，或者指向一个相关的表格；表格中存放的不是虚基类子对象的地址，就是其偏移量，此类指针被称为vbptr(虚基类指针)，如上图所示。如果既存在vptr又存在vbptr，某些编译器会将其优化，合并为一个指针。

指针的指向是从bc指向a的地址的

若为多继承时，内存布局如下：



虚继承的内存布局如下：



B中的偏移量+20=C中的偏移量+12 = 20

虚基类指针vbptr指向虚基类表vbtable，后者存放相对于当前位置的偏移量，避免多次继承A

19.什么情况下自动生成默认构造？

1.基类有默认构造，子类会合成构造调用并且调用基类构造

2.带有一个虚函数或者虚基类的类

3.合成的默认构造中，只有基类子对象和成员类对象会被初始化，其他非静态数据成员不会被初始化。

4.一个类有成员对象，成员对象有默认构造，难么基类也会有，只不过要在构造函数真正被需要的时候才会合成。

20.抽象基类为什么不能创建对象？

1.抽象类：带有纯虚函数的类是抽象类。

2.作用：有关操作作为接口，为派生类提供一个公共的根，派生类具体再去实现。类似于接口。

3.抽象类只能作为基类来使用，纯虚函数的由派生类实现。如果不实现，那这个类也是一个抽象类。

4.纯虚函数定义：一种特殊的虚函数 =0；

5.引入：

- 方便多态特性的使用
- 很多情况下，基类本身生成对象是不合理的，动物！=老虎狮子

6.相似概念：

- 多态性：相同对象收到不同消息或者不同对象收到相同消息是产生的不同的实现动作，包括编译时多态和运行时多态
- 编译时多态：重载 运行时多态：虚函数
- 虚函数
- 抽象类

21.模板类和模板函数的区别？

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。即函数模板允许隐式调用和显式调用而类模板只能显示调用。在使用时类模板必须加，而函数模板不必

22.多继承的优点？

1.C++一个派生类指定多个基类，叫做多重继承

2.优点是调用多个接口

3.如果继承的多个基类有共同的基类，且派生类对象需要调用祖先类的接口，容易产生二义性

4.可以加上全局符确定调用哪一份拷贝。比如pa.Author::eat()调用属于Author的拷贝。

5.使用虚拟继承，让多重继承类只拥有祖先类的一份拷贝。

23.模板和实现能够不写在一个文件里？为什么？

因为在编译时模板并不能生成真正的二进制代码，而是在编译调用模板类或函数的CPP文件时才会去找对应的模板声明和实现，在这种情况下编译器是不知道实现模板类或函数的CPP文件的存在，所以它只能找到模板类或函数的声明而找不到实现，而只好创建一个符号寄希望于链接程序找地址。但模板类或函数的实现并不能被编译成二进制代码，结果链接程序找不到地址只好报错了。

为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

24.将字符串hello world从开始到打印到屏幕上的全过程？

1.用户告诉操作系统执行HelloWorld程序（通过键盘输入等）

2．操作系统：找到helloworld程序的相关信息，检查其类型是否是可执行文件；并通过程序首部信息，确定代码和数据在可执行文件中的位置并计算出对应的磁盘块地址。

3．操作系统：创建一个新进程，将HelloWorld可执行文件映射到该进程结构，表示由该进程执行helloworld程序。

4．操作系统：为helloworld程序设置cpu上下文环境，并跳到程序开始处。

5．执行helloworld程序的第一条指令，发生缺页异常

6．操作系统：分配一页物理内存，并将代码从磁盘读入内存，然后继续执行helloworld程序

7．helloword程序执行puts函数（系统调用），在显示器上写一字符串

8．操作系统：找到要将字符串送往的显示设备，通常设备是由一个进程控制的，所以，操作系统将要写的字符串送给该进程

9．操作系统：控制设备的进程告诉设备的窗口系统，它要显示该字符串，窗口系统确定这是一个合法的操作，然后将字符串转换成像素，将像素写入设备的存储映像区

10．视频硬件将像素转换成显示器可接收和一组控制数据信号

11．显示器解释信号，激发液晶屏

12．OK，我们在屏幕上看到了HelloWorld

25.为什么拷贝构造函数必须传引用不能传值？

会无限递归，内存溢出。之前有讲过。

26.静态函数能定义为虚函数吗？常函数呢？说说你的理解

1、static成员不属于任何类对象或类实例，所以即使给此函数加上virutal也是没有任何意义的。

2、静态与非静态成员函数之间有一个主要的区别，那就是静态成员函数没有this指针。

虚函数依靠vptr和vtable来处理。vptr是一个指针，在类的构造函数中创建生成，并且只能用this指针来访问它，因为它是类的一个成员，并且vptr指向保存虚函数地址的vtable.对于静态成员函数，它没有this指针，所以无法访问vptr。

这就是为何static函数不能为virtual，虚函数的调用关系：this -> vptr -> vtable ->virtual function

27.虚函数的代价

1.带有虚函数的类，每一个类会产生一个虚函数表，用来存储指向虚成员函数的指针，会增大类

2.带有虚函数的类的每一个对象，都会有一个指向虚表的指针，增加对象的空间大小

3.不能再是内联的函数，因为内联函数在编译阶段进行替代，虚函数表示等待，在运行阶段才能确定调用哪种函数。

28.说一下移动构造函数

1.初始化后的对象不再使用了，浪费了可以，可以避免新空间的分配，降低构造成本，这是设计初衷。

2.拷贝构造，指针要深拷贝，移动构造，指针要浅拷贝

3.不是复制，就是将不使用的临时变量本身移动到目的对象。使用一个临时变量对象进行构造初始化的时候，就要调用移动构造函数。

29.那什么时候需要合成拷贝构造函数呢？

有三种情况会以一个对象的内容作为另一个对象的初值：

1. 对一个对象做显示的初始化操作，X xx = x;
2. 当对象被当做参数交给某个函数时；
3. 当函数传回一个类对象时；

如果一个类没有拷贝构造函数

但是有一个类类型的成员变量，它有拷贝构造函数

该类继承自有拷贝构造函数的基类

该类声明或继承了虚函数

该类中有虚基类

以上四种情况都会为该类合成一个拷贝构造函数

30.构造函数的执行顺序

1.在派生类构造函数中，所有的虚基类及上一层基类的构造函数调用；

2.对象的vptr被初始化；

3.如果有成员初始化列表，将在构造函数体内扩展开来，这必须在vptr被设定之后才做；

4.执行程序员所提供的代码；

31.哪些函数不能是虚函数？把你知道的都说一说

1.构造函数：构造初始化对象，派生类必须知道基函数干了啥才能构造

2.内联函数：要在编译阶段展开，虚函数在运行中才能确定

3.静态函数：不属于对象属于类，没有this指针，访问不了vptr，设为虚函数没意义

4.友元函数：不属于类的成员函数，无法被继承

5.普通函数，同上

32.什么是纯虚函数，与虚函数的区别

虚函数是为了实现动态编联产生的，目的是通过基类类型的指针指向不同对象时，自动调用相应的、和基类同名的函数（使用同一种调用形式，既能调用派生类又能调用基类的同名函数）。虚函数需要在基类中加上virtual修饰符修饰，因为virtual会被隐式继承，所以子类中相同函数都是虚函数。当一个成员函数被声明为虚函数之后，其派生类中同名函数自动成为虚函数，在派生类中重新定义此函数时要求函数名、返回值类型、参数个数和类型全部与基类函数相同。

纯虚函数只是相当于一个接口名，但含有纯虚函数的类不能够实例化。