语法 是 很不重要的，基本的回会了就行了

商汤科技 cpp 面经

1.程序的内存布局?--可以详看施磊老师第一节课

布局大概	
.text(代码段,放指令), .rodata(只读数据段,比如:常量字符串)---只读,不写 .data(数据段: 存放初始化的,且初始化不为0的) .dss(数据段: 存放未初始化的,和初始化为0的) .heap(堆)(只有运行new了才有, 低地址向高地址) .so, .dll(共享库,静态与动态) stack(栈空间, 从下往上增长,高地址向低地址) 命令行参数和环境变量	0x00000000 0x08048000 0xc0000000 3G-----用户空间
内核区(kernel space) **ZONE_DMA**: 前 16MB，用于支持老旧设备的 DMA。 **ZONE_DMA32**（仅在 64 位系统中存在）：支持 32 位地址的 DMA，范围是前 4GB。 **ZONE_NORMAL**: 常规内存区域，通常用于内核和用户空间的普通内存分配。 **ZONE_HIGHMEM**（仅在 32 位系统中存在）：用于映射超过 1GB 的物理内存（仅在 32 位系统中需要）。	1G---内核空间--0xffffffff

```
0x00000000:

这是一个空指针地址，通常用于表示无效或未初始化的指针。

在许多操作系统中，这个地址是保留的，访问它可能会导致段错误（segmentation fault）或程序崩溃。

0x08048000:

这是32位x86架构的Linux可执行文件中，.text段（存放可执行代码的部分）的默认起始地址。

它是Linux系统中使用ld链接器生成的可执行文件的默认基地址。
```

1. **数据段** 被称为 **静态内存区** 是很**不专业**的术语!!!
2. text和rodata是在一起吗?
   并不在一块存，不同的页面上，从内存的属性上,即只读的，可以理解为一块内存

2.堆栈区别

1. 内存方面的：堆内存和栈内存

   malloc或者new调用堆内存，free和delete释放堆内存，手动开辟和释放
   栈内存在调用函数就会占用栈内存，系统自动开辟和释放
2. 数据结构方面的：二叉堆,大根堆,小根堆， 栈?
   栈是一个先进后出的线性表，堆常用的二叉堆，就是一个二叉树，二叉堆里经常用的就是大根堆(堆顶是最大的)和小根堆(堆顶是最小的)
   priority_queue 就是默认用大根堆实现的
   栈 stack

3.函数调用参数是怎么传递的?

1. 实参传给 形参---->太捞了
2. 讲讲函数 调用的 压栈行为
   多参数，**从右往左压栈**，压完参数，**压调用方的下一行指令地址**(为了在函数执行完毕后，程序知道从哪里继续执行)，**然后把调用方的栈底地址ebp压上**(这是为了在函数返回时能够恢复调用方的栈帧)，**然后ebp指针指向这里的esp作为**

新的栈底，开始压调用的函数内容使用esp指针偏移进行后续的压栈，**使用ebp偏移访问形参**

4.为什么函数调用从右往左压栈

1. 因为c/cpp是为了支持可变参数

```
指令是在 编译时生成的，编译阶段 是无法知道到底有多少 可变参数的，需要把确定的参数，放到离 ebp 最近的地方，编译器永远知道，ebp+4是第一个参数
从右往左压参数，可以保证第一个参数，就在ebp+4的地方


```

5.函数题

```
string fun(string s1, string s2)
{
    string tmp = s1+s2;
    return tmp;
}
```

1. 主函数通过 string s = fun(s1+s2); 调用，依照代码执行顺序分析一下调用了什么 构造函数和顺序，以及析构函数的调用顺序

```
 1. 因为是值传递，实参s2到形参s2的拷贝构造，实参s1到形参s1的拷贝构造
2. s1+s2到tmp的拷贝构造
3. string s = fun(s1+s2) ，fun(s1+s2)返回来的并不产生临时对象， 因此变为了 string s = "hello"； 这是一个拷贝构造
4.被调用的函数里面开始析构了，析构 tmp,s1，s2，注意顺序
5.最后主函数完了后，析构s


知识点：任意cpp编译器 都会做优化，什么优化？
如果用临时对象 拷贝构造 新对象，那么临时对象就不产生了，直接构造新对象
临时对象是指在表达式求值过程中由编译器自动创建的对象。这些对象通常没有显式的名字，生命周期短暂，仅在表达式求值期间存在，之后会被销毁
```

2. 如果 return s1+s2; 与原来有什么区别？

```
省略了 tmp的拷贝构造和析构函数
```

3. 优化

```
1. 参数传递使用引用，省略拷贝构造
2. 返回对象时,直接返回结果，不要先定义再返回
```

6.类和结构体的内存对齐----空结构体

```
struct Data{
    char a;
    double b;
}   //2*8=16字节

struct Data{
    char a;  //1字节, 8bits
    char b;
    char c:
}   //3字节
```

重点：空结构体

```
struct Data{

}

win下面, vs 的.c 文件, 不允许定义空结构体

gcc linux下, 空结构体是 0字节

win下面 vs 和 gcc/g++ linux下, .cpp .cc cpp语言 是 1字节
```

为什么?

```
在c里, struct 是变量, 叫结构体变量

 cpp里 struct不是变量, 叫对象

变量只需要内存,没有东西, 就是0
对象不仅有对象, 还有构造, 而且构造函数 会生成this指针 ,  内存最小单位是 1字节, 所以是1字节
this 指针是一个隐含的指针，指向当前对象的地址。它的确占用 4 字节（32 位系统）或 8 字节（64 位系统），但 this 指针本身并不是对象的一部分，而是编译器在调用成员函数时隐式传递的一个参数。


this 指针的本质
```

```
this 指针是一个隐式参数，它在调用成员函数时由编译器自动传递。具体来说：

当你调用一个成员函数时，编译器会将当前对象的地址作为第一个参数传递给该函数。

这个地址就是 this 指针的值。


```

cpp中， 只有一个字节的变量的对象， 也是一字节大小

```
基类是 空， 派生类本身也是空
则 派生类也是1,因为啥也没继承， 而不要考虑 基类+派生类=2
那如果虚继承类呢， 会有vbptr， 则派生类将是 4 字节(指针大小)， 若还有虚函数， 将还有vfptr， 将是8字节
```

7.智能指针

防止内存泄漏,资源泄露
不带引用计数的智能指针： **auto_ptr, unique_ptr**
带引用计数的智能指针:**shared_ptr, weak_ptr**

若使用智能指针shared_ptr， 会造成交叉引用问题， 导致资源无法释放----详见高级课程