拓跋阿秀-C++面试八股

2.1.1、基础语法

1、在 main 执行之前和之后执行的代码可能是什么?

main 函数执行之前，主要就是初始化系统相关资源：

- 设置栈指针
- 初始化静态 `static` 变量和 `global` 全局变量，即`.data`段的内容
- 将未初始化部分的全局变量赋初值：数值型 `short`，`int`，`long` 等为 `0`，`bool` 为 `FALSE`，指针为 `NULL` 等等，即`.bss`段的内容
- 全局对象初始化，在 `main` 之前调用构造函数，这是可能会执行前的一些代码
- 将 main 函数的参数 `argc`，`argv` 等传递给 `main` 函数，然后才真正运行 `main` 函数
- `__attribute__((constructor))`

main 函数执行之后：

- 全局对象的析构函数会在 main 函数之后执行;
- 可以用 `atexit` 注册一个函数，它会在 main 之后执行;
- `__attribute__((destructor))`

update1:https://github.com/forthespada/InterviewGuide/issues/2 ,由 `stanleyguo0207` 提出 - 2021.03.22

2、结构体内存对齐问题?

- 结构体内成员按照声明顺序存储，第一个成员地址和整个结构体地址相同。
- 未特殊说明时，按结构体中 size 最大的成员对齐（若有 double 成员，按 8 字节对齐。）

c++11 以后引入两个关键字 alignas 与 alignof。其中 `alignof` 可以计算出类型的对齐方式，`alignas` 可以指定结构体的对齐方式。

但是 `alignas` 在某些情况下是不能使用的，具体见下面的例子:

```
Plaintext
```

```
// alignas 生效的情况

struct Info {
  uint8_t a;
  uint16_t b;
  uint8_t c;
};

std::cout << sizeof(Info) << std::endl;   // 6  2 + 2 + 2
std::cout << alignof(Info) << std::endl;  // 2

struct alignas(4) Info2 {
  uint8_t a;
  uint16_t b;
  uint8_t c;
};

std::cout << sizeof(Info2) << std::endl;   // 8  4 + 4
std::cout << alignof(Info2) << std::endl;  // 4
```

`alignas` 将内存对齐调整为 4 个字节。所以 `sizeof(Info2)`的值变为了 8。

```
Plaintext
// alignas 失效的情况

struct Info {
  uint8_t a;
  uint32_t b;
  uint8_t c;
};

std::cout << sizeof(Info) << std::endl;   // 12  4 + 4 + 4
std::cout << alignof(Info) << std::endl;  // 4

struct alignas(2) Info2 {
  uint8_t a;
  uint32_t b;
  uint8_t c;
};

std::cout << sizeof(Info2) << std::endl;   // 12  4 + 4 + 4
std::cout << alignof(Info2) << std::endl;  // 4
```

若 `alignas` 小于自然对齐的最小单位，则被忽略。

- 如果想使用单字节对齐的方式，使用 `alignas` 是无效的。应该使用`#pragma pack(push,1)`或者使用`__attribute__((packed))`。

```
Plaintext
#if defined(__GNUC__) || defined(__GNUG__)
  #define ONEBYTE_ALIGN __attribute__((packed))
#elif defined(_MSC_VER)
  #define ONEBYTE_ALIGN
  #pragma pack(push,1)
#endif

struct Info {
  uint8_t a;
  uint32_t b;
  uint8_t c;
} ONEBYTE_ALIGN;

#if defined(__GNUC__) || defined(__GNUG__)
  #undef ONEBYTE_ALIGN
#elif defined(_MSC_VER)
  #pragma pack(pop)
  #undef ONEBYTE_ALIGN
#endif

std::cout << sizeof(Info) << std::endl;   // 6 1 + 4 + 1
std::cout << alignof(Info) << std::endl;  // 1
```

- 确定结构体中每个元素大小可以通过下面这种方法:

```
Plaintext
#if defined(__GNUC__) || defined(__GNUG__)
  #define ONEBYTE_ALIGN __attribute__((packed))
#elif defined(_MSC_VER)
  #define ONEBYTE_ALIGN
  #pragma pack(push,1)
#endif

/**
* 0 1   3     6   8 9            15
* +-+---+-----+---+-+-------------+
* | |   |     |   | |             |
* |a| b |  c  | d |e|     pad     |
* | |   |     |   | |             |
* +-+---+-----+---+-+-------------+
```

```
*/
struct Info {
  uint16_t a : 1;
  uint16_t b : 2;
  uint16_t c : 3;
  uint16_t d : 2;
  uint16_t e : 1;
  uint16_t pad : 7;
} ONEBYTE_ALIGN;

#if defined(__GNUC__) || defined(__GNUG__)
  #undef ONEBYTE_ALIGN
#elif defined(_MSC_VER)
  #pragma pack(pop)
  #undef ONEBYTE_ALIGN
#endif

std::cout << sizeof(Info) << std::endl;   // 2
std::cout << alignof(Info) << std::endl;  // 1
```

- 这种处理方式是处理不了的。

update1:https://github.com/forthespada/InterviewGuide/issues/2 ,由 `stanleyguo0207` 提出 - 2021.03.22

3、指针和引用的区别

- 指针是一个变量，存储的是一个地址，引用跟原来的变量实质上是同一个东西，是原变量的别名
- 指针可以有多级，引用只有一级
- 指针可以为空，引用不能为 NULL 且在定义时必须初始化
- 指针在初始化后可以改变指向，而引用在初始化之后不可再改变
- sizeof 指针得到的是本指针的大小，sizeof 引用得到的是引用所指向变量的大小
- 当把指针作为参数进行传递时，也是将实参的一个拷贝传递给形参，两者指向的地址相同，但不是同一个变量，在函数中改变这个变量的指向不影响实参，而引用却可以。
- 引用本质是一个指针，同样会占 4 字节内存；指针是具体变量，需要占用存储空间（，具体情况还要具体分析）。
- 引用在声明时必须初始化为另一变量，一旦出现必须为 typename refname &varname 形式；指针声明和定义可以分开，可以先只声明指针变量而不初始化，等

用到时再指向具体变量。

- 引用一旦初始化之后就不可以再改变（变量可以被引用为多次，但引用只能作为一个变量引用）；指针变量可以重新指向别的变量。
- 不存在指向空值的引用，必须有具体实体；但是存在指向空值的指针。

参考代码：

```
Plaintext
void test(int *p)
{
    int a=1;
    p=&a;
    cout<<p<<" "<<*p<<endl;
}

int main(void)
{
    int *p=NULL;
    test(p);
    if(p==NULL)
    cout<<"指针 p 为 NULL"<<endl;
    return 0;
}
//运行结果为：
//0x22ff44 1
//指针 p 为 NULL


void testPTR(int* p) {
        int a = 12;
        p = &a;

}

void testREFF(int& p) {
        int a = 12;
        p = a;

}
void main()
{
        int a = 10;
        int* b = &a;
```

```
        testPTR(b);//改变指针指向，但是没改变指针的所指的内容
        cout << a << endl;// 10
        cout << *b << endl;// 10

        a = 10;
        testREFF(a);
        cout << a << endl;//12
}
```

在编译器看来, int a = 10; int &b = a; 等价于 int * const b = &a; 而 b = 20; 等价于 *b = 20; 自动转换为指针和自动解引用.

4、在传递函数参数时，什么时候该使用指针，什么时候该使用引用呢?

- 需要返回函数内局部变量的内存的时候用指针。使用指针传参需要开辟内存，用完要记得释放指针，不然会内存泄漏。而返回局部变量的引用是没有意义的
- 对栈空间大小比较敏感（比如递归）的时候使用引用。使用引用传递不需要创建临时变量，开销要更小
- 类对象作为参数传递的时候使用引用，这是 C++类对象传递的标准方式

5、堆和栈的区别

- 申请**方式**不同。
  - 栈由系统自动分配。
  - 堆是自己申请和释放的。
- 申请**大小限制**不同。
  - 栈顶和栈底是之前预设好的，栈是向栈底扩展，大小固定，可以通过 ulimit -a 查看，由 ulimit -s 修改。
  - 堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。
- 申请**效率**不同。
  - 栈由系统分配，速度快，不会有碎片。
  - 堆由程序员分配，速度慢，且会有碎片。

栈空间默认是 4M, 堆区一般是 1G - 4G

	堆	栈

管理方式	堆中资源由程序员控制（容易产生 memory leak）	栈资源由编译器自动管理，无需手工控制
内存管理机制	系统有一个记录空闲内存地址的链表，当系统收到程序申请时，遍历该链表，寻找第一个空间大于申请空间的堆结点，删除空闲结点链表中的该结点，并将该结点空间分配给程序（大多数系统会在这块内存空间首地址记录本次分配的大小，这样 delete 才能正确释放本内存空间，另外系统会将多余的部分重新放入空闲链表中）	只要栈的剩余空间大于所申请空间，系统为程序提供内存，否则报异常提示栈溢出。（这一块理解一下链表和队列的区别，不连续空间和连续空间的区别，应该就比较好理解这两种机制的区别了）
空间大小	堆是不连续的内存区域（因为系统是用链表来存储空闲内存地址，自然不是连续的），堆大小受限于计算机系统中有效的虚拟内存（32bit 系统理论上是 4G），所以堆的空间比较灵活，比较大	栈是一块连续的内存区域，大小是操作系统预定好的，windows 下栈大小是 2M（也有是 1M，在编译时确定，VC 中可设置）
碎片问题	对于堆，频繁的 new/delete 会造成大量碎片，使程序效率降低	对于栈，它是有点类似于数据结构上的一个先进后出的栈，进出一一对应，不会产生碎片。（看到这里我突然明白了为什么面试官在问我堆和栈的区别之前先问了我栈和队列的区别）
生长方向	堆向上，向高地址方向增	栈向下，向低地址方向增

	长。	长。
分配方式	堆都是动态分配（没有静态分配的堆）	栈有静态分配和动态分配，静态分配由编译器完成（如局部变量分配），动态分配由 alloca 函数分配，但栈的动态分配的资源由编译器进行释放，无需程序员实现。
分配效率	堆由 C/C++函数库提供，机制很复杂。所以堆的效率比栈低很多。	栈是其系统提供的数据结构，计算机在底层对栈提供支持，分配专门 寄存器存放栈地址，栈操作有专门指令。

形象的比喻

栈就像我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

6、你觉得堆快一点还是栈快一点？

毫无疑问是栈快一点。

因为操作系统会在底层对栈提供支持，会分配专门的寄存器存放栈的地址，栈的入栈出栈操作也十分简单，并且有专门的指令执行，所以栈的效率比较高也比较快。

而堆的操作是由 C/C++函数库提供的，在分配堆内存的时候需要一定的算法寻找合适大小的内存。并且获取堆的内容需要两次访问，第一次访问指针，第二次根据指针保存的地址访问内存，因此堆比较慢。

7、区别以下指针类型？

```
Plaintext
int *p[10]
int (*p)[10]
int *p(int)
int (*p)(int)
```

- int* p[10]表示指针数组，强调数组概念，是一个数组变量，数组大小为 10，数组内每个元素都是指向 int 类型的指针变量。
- int (*p)[10]表示数组指针，强调是指针，只有一个变量，是指针类型，不过指向的是一个 int 类型的数组，这个数组大小是 10。
- int* p(int)是函数声明，函数名是 p，参数是 int 类型的，返回值是 int *类型的。
- int (*p)(int)是函数指针，强调是指针，该指针指向的函数具有 int 类型参数，并且返回值是 int 类型的。

8、new / delete 与 malloc / free 的异同

相同点

- 都可用于内存的动态申请和释放

不同点

- 前者是 C++运算符，后者是 C/C++语言标准库函数
- new 自动计算要分配的空间大小，malloc 需要手工计算
- new 是类型安全的，malloc 不是。例如：

```
Plaintext
int *p = new float[2]; //编译错误
int *p = (int*)malloc(2 * sizeof(double));//编译无错误 前面必须强转
```

- new 调用名为 **operator new** 的标准库函数分配足够空间并调用相关对象的构造函数，delete 对指针所指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。后者均没有相关调用
- 后者需要库文件支持，前者不用
- new 是封装了 malloc，直接 free 不会报错，但是这只是释放内存，而不会析构对象

9、new 和 delete 是如何实现的?

- 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

10、malloc 和 new 的区别?

- malloc 和 free 是标准库函数，支持覆盖；new 和 delete 是运算符，不重载。
- 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；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 和 free 返回的是 void 类型指针(void*)（必须进行类型转换），new 和 delete 返回的是具体类型指针。

update1:感谢微信好友“猿六学算法”指出错误，已修正！

11、既然有了 malloc/free，C++中为什么还需要 new/delete 呢？直接用 malloc/free 不好吗？

- malloc/free 和 new/delete 都是用来申请内存和回收内存的。
- 在对非基本数据类型的对象使用的时候，对象创建的时候还需要执行构造函数，销毁的时候要执行析构函数。而 malloc/free 是库函数，是已经编译的代码，所以不能把构造函数和析构函数的功能强加给 malloc/free，所以 new/delete 是必不可少的。

12、被 free 回收的内存是立即返还给操作系统吗？

不是的，被 free 回收的内存会首先被 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。

13、宏定义和函数有何区别？

- 宏在编译时完成替换，之后被替换的文本参与编译，相当于直接插入了代码，运行时不存在函数调用，执行起来更快；函数调用在运行时需要跳转到具体调用函数。
- 宏定义属于在结构中插入代码，没有返回值；函数调用具有返回值。
- 宏定义参数没有类型，不进行类型检查；函数参数具有类型，需要检查类型。
- 宏定义不要在最后加分号。

14、宏定义和 typedef 区别？

- 宏主要用于定义常量及书写复杂的内容；typedef 主要用于定义类型别名。
- 宏替换发生在编译阶段之前，属于文本插入替换；typedef 是编译的一部分。
- 宏不检查类型；typedef 会检查数据类型。
- 宏不是语句，不在在最后加分号；typedef 是语句，要加分号标识结束。

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

15、变量声明和定义区别?

- 声明仅仅是把变量的声明的位置及类型提供给编译器，并不分配内存空间；定义要在定义的地方为其分配存储空间。
- 相同变量可以在多处声明（外部变量 extern），但只能在一处定义。

16、strlen 和 sizeof 区别?

- sizeof 是运算符，并不是函数，结果在编译时得到而非运行中获得；strlen 是字符处理的库函数。
- sizeof 参数可以是任何数据的类型或者数据（sizeof 参数不退化）；strlen 的参数只能是字符指针且结尾是'\0'的字符串。
- 因为 sizeof 值在编译时确定，所以不能用来得到动态分配（运行时分配）存储空间的大小。

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

      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
      return 0;
  }
```

17、常量指针和指针常量区别?

- 指针常量是一个指针，读成常量的指针，指向一个只读变量。如 int const *p 或 const int *p。
- 常量指针是一个不能给改变指向的指针。指针是个常量，不能中途改变指向，如 int *const p。
- update1:https://www.nowcoder.com/discuss/597948 ， 网友“ 牛客 191489444 号 ”指出笔误，感谢!

18、a 和&a 有什么区别?

```
C++
假设数组
```

```
int a[10];
int (*p)[10] = &a;
```

- a 是数组名，是数组首元素地址，+1 表示地址值加上一个 int 类型的大小，如果 a 的值是 0x00000001，加 1 操作后变为 0x00000005。*(a + 1) = a[1]。
- &a 是数组的指针，其类型为 int (*)[10]（就是前面提到的数组指针），其加 1 时，系统会认为是数组首地址加上整个数组的偏移（10 个 int 型变量），值为数组 a 尾元素后一个元素的地址。
- 若(int *)p，此时输出 *p 时，其值为 a[0]的值，因为被转为 int *类型，解引用时按照 int 类型大小来读取。

19、C++和 Python 的区别

包括但不限于：

- Python 是一种脚本语言，是解释执行的，而 C++是编译语言，是需要编译后在特定平台运行的。python 可以很方便的跨平台，但是效率没有 C++高。
- Python 使用缩进来区分不同的代码块，C++使用花括号来区分
- C++中需要事先定义变量的类型，而 Python 不需要，Python 的基本数据类型只有数字，布尔值，字符串，列表，元组等等
- Python 的库函数比 C++的多，调用起来很方便

20、C++和 C 语言的区别

- C++中 new 和 delete 是对内存分配的运算符，取代了 C 中的 malloc 和 free。
- 标准 C++中的字符串类取代了标准 C 函数库头文件中的字符数组处理函数（C 中没有字符串类型）。
- C++中用来做控制态输入输出的 iostream 类库替代了标准 C 中的 stdio 函数库。
- C++中的 try/catch/throw 异常处理机制取代了标准 C 中的 setjmp()和 longjmp()函数。
- 在 C++中，允许有相同的函数名，不过它们的参数类型不能完全相同，这样这些函数就可以相互区别开来。而这在 C 语言中是不允许的。也就是 C++可以重载，C 语言不允许。
- C++语言中，允许变量定义语句在程序中的任何地方，只要在是使用它之前就可以；而 C 语言中，必须要在函数开头部分。而且 C++允许重复定义变量，C 语言也是做不到这一点的
- 在 C++中，除了值和指针之外，新增了引用。引用型变量是其他变量的一个别名，

我们可以认为他们只是名字不相同，其他都是相同的。

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

21、C++与 Java 的区别

语言特性

- Java 语言给开发人员提供了更为简洁的语法；完全面向对象，由于 JVM 可以安装到任何的操作系统上，所以说它的可移植性强
- Java 语言中没有指针的概念，引入了真正的数组。不同于 C++中利用指针实现的“伪数组”，Java 引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在 C++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题
- C++也可以在其他系统运行，但是需要不同的编码（这一点不如 Java，只编写一次代码，到处运行），例如对一个数字，在 windows 下是大端存储，在 unix 中则为小端存储。Java 程序一般都是生成字节码，在 JVM 里面运行得到结果
- Java 用接口(Interface)技术取代 C++程序中的抽象类。接口与抽象类有同样的功能，但是省却了在实现和维护上的复杂性

垃圾回收

- C++用析构函数回收垃圾，写 C 和 C++程序时一定要注意内存的申请和释放
- Java 语言不使用指针，内存的分配和回收都是自动进行的，程序员无须考虑内存碎片的问题

应用场景

- Java 在桌面程序上不如 C++实用，C++可以直接编译成 exe 文件，指针是 c++的优势，可以直接对内存的操作，但同时具有危险性 。（操作内存的确是一项非常危险的事情，一旦指针指向的位置发生错误，或者误删除了内存中某个地址单元存放的重要数据，后果是可想而知的）
- Java 在 Web 应用上具有 C++ 无可比拟的优势，具有丰富多样的框架
- 对于底层程序的编程以及控制方面的编程，C++很灵活，因为有句柄的存在

update1:微信好友“宇少”进行“多继承”->"抽象类"的勘误

22、C++中 struct 和 class 的区别

相同点

- 两者都拥有成员函数、公有和私有部分

- 任何可以使用 class 完成的工作，同样可以使用 struct 完成

不同点

- 两者中如果不对成员不指定公私有，struct 默认是公有的，class 则默认是私有的
- class 默认是 private 继承，而 struct 模式是 public 继承

引申：C++和 C 的 struct 区别

- C 语言中：struct 是用户自定义数据类型（UDT）；C++中 struct 是抽象数据类型（ADT），支持成员函数的定义，（C++中的 struct 能继承，能实现多态）
- C 中 struct 是没有权限的设置的，且 struct 中只能是一些变量的集合体，可以封装数据却不可以隐藏数据，而且成员**不可以是函数**
- C++中，struct 增加了访问权限，且可以和类一样有成员函数，成员默认访问说明符为 public（为了与 C 兼容）
- struct 作为类的一种特例是用来自定义数据结构的。一个结构标记声明后，在 C 中必须在结构标记前加上 struct，才能做结构类型名（除：typedef struct class{};）;C++中结构体标记（结构体名）可以直接作为结构体类型名使用，此外结构体 struct 在 C++中被当作类的一种特例

23、define 宏定义和 const 的区别

编译阶段

- define 是在编译的**预处理**阶段起作用，而 const 是在编译、运行的时候起作用

安全性

- define 只做替换，不做类型检查和计算，也不求解，容易产生错误，一般最好加上一个大括号包含住全部的内容，要不然很容易出错
- const 常量有数据类型，编译器可以对其进行类型安全检查

内存占用

- define 只是将宏名称进行替换，在内存中会产生多分相同的备份。const 在程序运行中只有一份备份，且可以执行常量折叠，能将复杂的的表达式计算出结果放入常量表
- 宏替换发生在编译阶段之前，属于文本插入替换；const 作用发生于编译过程中。
- 宏不检查类型；const 会检查数据类型。
- 宏定义的数据没有分配内存空间，只是插入替换掉；const 定义的变量只是值不能改变，但要分配内存空间。

24、C++中 const 和 static 的作用

static

- 不考虑类的情况
  - 隐藏。所有不加 static 的全局变量和函数具有全局可见性，可以在其他文件中使用，加了之后只能在该文件所在的编译模块中使用
  - 默认初始化为 0，包括未初始化的全局静态变量与局部静态变量，都存在全局未初始化区
  - 静态变量在函数内定义，始终存在，且只进行一次初始化，具有记忆性，其作用范围与局部变量相同，函数退出后仍然存在，但不能使用
- 考虑类的情况
  - static 成员变量：只与类关联，不与类的对象关联。定义时要分配空间，不能在类声明中初始化，必须在类定义体外部初始化，初始化时不需要标示为 static；可以被非 static 成员函数任意访问。
  - static 成员函数：不具有 this 指针，无法访问类对象的非 static 成员变量和非 static 成员函数；**不能被声明为 const、虚函数和 volatile**；可以被非 static 成员函数任意访问

const

- 不考虑类的情况
  - const 常量在定义时必须初始化，之后无法更改
  - const 形参可以接收 const 和非 const 类型的实参，例如

```
C
// i 可以是 int 型或者 const int 型 void fun(const int&
i){        //...}
```

- 考虑类的情况
  - const 成员变量：不能在类定义外部初始化，只能通过构造函数初始化列表进行初始化，并且必须有构造函数；不同类对其 const 数据成员的值可以不同，所以不能在类中声明时初始化
  - const 成员函数：const 对象不可以调用非 const 成员函数；非 const 对象都可以调用；不可以改变非 mutable（用该关键字声明的变量可以在 const 成员函数中被修改）数据的值

25、C++的顶层 const 和底层 const

区分指针常量和常量指针

中文翻译过来的这两个概念非常容易混淆，直接用定义语句记：

概念区分

- **顶层** const：指的是 const 修饰的**本身**是一个常量，无法修改，指的是指针，就是 * 号的右边
- **底层** const：指的是 const 修饰的变量**所指向的对象**是一个常量，指的是所指变量，就是 * 号的左边

举个例子

```
C++
int a = 10;
int* const b1 = &a;
//顶层 const，b1 本身是一个常量 const int* b2 = &a;
//底层 const，b2 本身可变，所指的对象是常量 const int b3 = 20;
//顶层 const，b3 是常量不可变 const int* const b4 = &a;
//前一个 const 为底层，后一个为顶层，b4 不可变 const int& b5 = a;
//用于声明引用变量，都是底层 const
```

区分作用

- 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

```
C++
const int a;
int const a;
const int *a;
int *const a;
```

- int const a 和 const int a 均表示定义常量类型 a。
- const int *a，其中 a 为指向 int 型变量的指针，const 在 * 左侧，表示 a 指向不可变常量。(看成 const (*a)，对引用加 const)
- int *const a，依旧是指针类型，表示 a 为指向整型数据的常指针。(看成 const(a)，对指针 const)

26、数组名和指针（这里为指向数组首元素的指针）区别?

- 二者均可通过增减偏移量来访问数组中的元素。

- 数组名不是真正意义上的指针，可以理解为常指针，所以数组名没有自增、自减等操作。
- **当数组名当做形参传递给调用函数后，就失去了原有特性，退化成一般指针，多了自增、自减操作，但 sizeof 运算符不能再得到原数组的大小了。**

27、final 和 override 关键字

override

当在父类中使用了虚函数时候，你可能需要在某个子类中对这个虚函数进行重写，以下方法都可以：

```
Plaintext
class A{    virtual void foo();}class B : public A{    void foo(); //OK    virtual void foo(); // OK    void foo() override; //OK}
```

如果不使用 override，当你手一抖，将**foo()写成了 f00()**会怎么样呢？结果是编译器并不会报错，因为它并不知道你的目的是重写虚函数，而是把它当成了新的函数。如果这个虚函数很重要的话，那就会对整个程序不利。所以，override 的作用就出来了，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：

```
Plaintext
class A{    virtual void foo();};class B : public A{    virtual void f00(); //OK，这个函数是 B 新增的，不是继承的    virtual void f0o() override; //Error，加了 override 之后，这个函数一定是继承自 A 的，A 找不到就报错};
```

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。例子如下：

```
Plaintext
class Base{    virtual void foo();}; class A : public Base{    void foo() final; // foo 被 override 并且是最后一个 override，在其子类中不可以重写};class B final : A // 指明 B 是不可以被继承的{    void foo() override; // Error: 在 A 中已经被 final 了};
class C : B // Error: B is final{};
```

28、拷贝初始化和直接初始化

- 当用于类类型对象时，初始化的拷贝形式和直接形式有所不同：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。举例如下

```
Plaintext
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 str3string 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
```

- 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
- （语句 1 和语句 3 等价），但是需要辨别两种情况。
  - 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

29、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
- 对于类和复杂数据类型来说，这两者的区别就大了，举例如下：

```
C++
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;}
```

30、extern"C"的用法

为了能够**正确的在 C++代码中调用 C 语言**的代码：在程序中加上 extern "C"后，相当于告诉编译器这部分代码是 C 语言写的，因此要按照 C 语言进行编译，而不是 C++；

哪些情况下使用 extern "C"：

（1）C++代码中调用 C 语言代码；

（2）在 C++中的头文件中使用；

（3）在多个人协同开发时，可能有人擅长 C 语言，而有人擅长 C++;

举个例子，C++中调用 C 代码:

```
C++
#ifndef __MY_HANDLE_H__#define __MY_HANDLE_H__extern
"C"{    typedef unsigned int result_t;    typedef void*
my_handle_t;        my_handle_t create_handle(const char* name);
result_t operate_on_handle(my_handle_t handle);    void
close_handle(my_handle_t handle);}
```

综上，总结出使用方法**，在 C 语言的头文件中，对其外部函数只能指定为 extern 类型，C 语言中不支持 extern "C"声明，在.c 文件中包含了 extern "C"时会出现编译语法错误。**所以使用 extern "C"全部都放在于 cpp 程序相关文件或其头文件中。

总结出如下形式:

（1）C++调用 C 函数:

```
Plaintext
//xx.hextern int add(...)//xx.cint add(){    }//xx.cppextern "C"
{    #include "xx.h"}
```

（2）C 调用 C++函数

```
Plaintext
//xx.hextern "C"{    int add();}//xx.cppint
add(){    }//xx.cextern int add();
```

31、野指针和悬空指针

都是是指向无效内存区域(这里的无效指的是"不安全不可控")的指针，访问行为将会导致未定义行为。

- 野指针

野指针，指的是没有被初始化过的指针

```
C++
int main(void) {        int* p;     // 未初始化    std::cout<< *p
<< std::endl; // 未初始化就被使用        return 0;}
```

- 因此，为了防止出错，对于指针初始化时都是赋值为 `nullptr`，这样在使用时编译器就会直接报错，产生非法内存访问。

- 悬空指针

悬空指针，指针最初指向的内存已经被释放了的一种指针。

```C++
int main(void) {
int * p = nullptr;
int* p2 = new int;
 p = p2;  delete p2;
}
```

此时 p 和 p2 就是悬空指针，指向的内存已经被释放。继续使用这两个指针，行为不可预料。需要设置为 `p=p2=nullptr`。此时再使用，编译器会直接保错。 避免野指针比较简单，但悬空指针比较麻烦。c++引入了智能指针，C++智能指针的本质就是避免悬空指针的产生。

产生原因及解决办法：

野指针：指针变量未及时初始化 => 定义指针变量及时初始化，要么置空。

悬空指针：指针 free 或 delete 之后没有及时置空 => 释放操作后立即置空。

32、C 和 C++的类型安全

什么是类型安全?

类型安全很大程度上可以等价于内存安全，类型安全的代码不会试图访问自己没被授权的内存区域。“类型安全”常被用来形容编程语言，其根据在于该门编程语言是否提供保障类型安全的机制；有的时候也用“类型安全”形容某个程序，判别的标准在于该程序是否隐含类型错误。

类型安全的编程语言与类型安全的程序之间，没有必然联系。好的程序员可以使用类型不那么安全的语言写出类型相当安全的程序，相反的，差一点儿的程序员可能使用类型相当安全的语言写出类型不太安全的程序。绝对类型安全的编程语言暂时还没有。

（1）C 的类型安全

C 只在局部上下文中表现出类型安全，比如试图从一种结构体的指针转换成另一种结构体的指针时，编译器将会报告错误，除非使用显式类型转换。然而，C 中相当多的操作是不安全的。以下是两个十分常见的例子：

- printf 格式输出

上述代码中，使用%d 控制整型数字的输出，没有问题，但是改成%f 时，明显输出错误，再改成%s 时，运行直接报 segmentation fault 错误

- malloc 函数的返回值

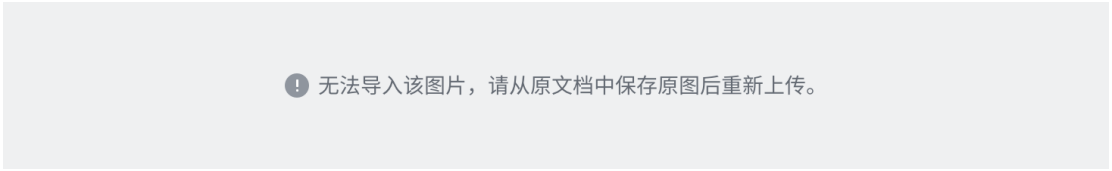
malloc 是 C 中进行内存分配的函数，它的返回类型是 void *即空类型指针，常常有这样的用法 char* pStr=(char*)malloc(100*sizeof(char))*，这里明显做了显式的类型转换。

类型匹配尚且没有问题，但是一旦出现 int* pInt=(int*)malloc(100*sizeof(char))就很可能带来一些问题，而这样的转换 C 并不会提示错误。

（2）C++的类型安全

如果 C++使用得当，它将远比 C 更有类型安全性。相比于 C 语言，C++提供了一些新的机制保障类型安全：

- 操作符 new 返回的指针类型严格与对象匹配，而不是 void*
- C 中很多以 void*为参数的函数可以改写为 C++模板函数，而模板是支持类型检查的；
- 引入 const 关键字代替#define constants，它是有类型、有作用域的，而#define constants 只是简单的文本替换
- 一些#define 宏可被改写为 inline 函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全
- C++提供了 **dynamic_cast** 关键字，使得转换过程更加安全，因为 dynamic_cast 比 static_cast 涉及更多具体的类型检查。
- 例 1：使用 void*进行类型转换



例 2：不同类型指针之间转换

```
C++
#include<iostream>
using namespace std;
class Parent{};
class Child1 :
public Parent{
public:
int i;
Child1(int e):i(e){}
};
class Child2 : public Parent{public:        double d;
Child2(double e):d(e){}};int main(){        Child1 c1(5);
Child2 c2(4.1);        Parent* pp;        Child1* pc1;
```

```
pp=&c1;          pc1=(Child1*)pp;  // 类型向下转换 强制转换，由于类型仍然为 Child1*，不造成错误          cout<<pc1->i<<endl; //输出：5
pp=&c2;         pc1=(Child1*)pp;  //强制转换，且类型发生变化，将造成错误         cout<<pc1->i<<endl;// 输出：1717986918         return 0;}
```

上面两个例子之所以引起类型不安全的问题，是因为程序员使用不得当。第一个例子用到了空类型指针 void*，第二个例子则是在两个类型指针之间进行强制转换。因此，想保证程序的类型安全性，应尽量避免使用空类型指针 void*，尽量不对两种类型指针做强制转换。

33、C++中的重载、重写（覆盖）和隐藏的区别

（1）重载（overload）

重载是指在同一范围定义中的同名成员函数才存在重载关系。主要特点是函数名相同，参数类型和数目有所不同，不能出现参数个数和类型均相同，仅仅依靠返回值不同来区分的函数。重载和函数成员是否是虚函数无关。举个例子：

```
Plaintext
class A{    ...    virtual int fun();    void fun(int);    void fun(double, double);    static int fun(char);    ...}
```

（2）重写（覆盖）（override）

重写指的是在派生类中覆盖基类中的同名函数，**重写就是重写函数体**，**要求基类函数必须是虚函数**且：

- 与基类的虚函数有相同的参数个数
- 与基类的虚函数有相同的参数类型
- 与基类的虚函数有相同的返回值类型

举个例子：

```
C++
//父类 class A{public:    virtual int fun(int a){}}//子类 class B : public A{public:    //重写,一般加 override 可以确保是重写父类的函数 virtual int fun(int a) override{}}
```

重载与重写的区别：

- 重写是父类和子类之间的垂直关系，重载是不同函数之间的水平关系
- 重写要求参数列表相同，重载则要求参数列表不同，返回值不要求
- 重写关系中，调用方法根据对象类型决定，重载根据调用时实参表与形参表的对

应关系来选择函数体

（3）隐藏（hide）

隐藏指的是某些情况下，派生类中的函数屏蔽了基类中的同名函数，包括以下情况：

- 两个函数参数相同，但是基类函数不是虚函数。**和重写的区别在于基类函数是否是虚函数。**举个例子：

```
C++
//父类 class A{public:    void fun(int a){                cout << "A 中的 fun 函数" << endl;        }};//子类 class B : public A{public:    //隐藏父类的 fun 函数    void fun(int a){                cout << "B 中的 fun 函数" << endl;        }};int main(){    B b;    b.fun(2); //调用的是 B 中的 fun 函数 b.A::fun(2); //调用 A 中 fun 函数    return 0;}
```

- **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。和重载的区别在于两个函数不在同一个类中。举个例子：**

```
C++
//父类 class A{public:    virtual void fun(int a){                cout << "A 中的 fun 函数" << endl;        }};//子类 class B : public A{public:    //隐藏父类的 fun 函数    virtual void fun(char* a){            cout << "A 中的 fun 函数" << endl;    }};int main(){    B b;    b.fun(2); //报错，调用的是 B 中的 fun 函数，参数类型不对    b.A::fun(2); //调用 A 中 fun 函数    return 0;}
```

34、C++有哪几种的构造函数

C++中的构造函数可以分为 4 类：

- 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

举个例子：

```
Plaintext
```

```
#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;};int main(){    Student s1;    Student s2(18,1001);    int a
= 10;    Student s3(a);    Student s4(s3);        printf("s1
age:%d, num:%d\n", s1.age, s1.num);    printf("s2 age:%d,
num:%d\n", s2.age, s2.num);    printf("s3 age:%d, num:%d\n",
s3.age, s3.num);    printf("s2 age:%d, num:%d\n", s4.age, s4.num);
return 0;}//运行结果//s1 age:20, num:1000//s2 age:18, num:1001//s3
age:10, num:1002//s2 age:10, num:1002
```

- 默认构造函数和初始化构造函数在定义类的对象，完成对象的初始化工作
- 复制构造函数用于复制本类的对象
- 转换构造函数用于将其他类型的变量，隐式转换为本类对象

35、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。

```
C++
#include <iostream>
#include <string.h>
using namespace std;
class Student{
private:
    int num;
    char *name;
public:
 Student(){
```

```
    name = new char(20);
    cout << "Student" << endl;
      };
  ~Student(){        cout << "~Student " << &name << endl;
delete name;        name = NULL;    };        Student(const
Student &s){//拷贝构造函数        //浅拷贝，当对象的 name 和传入对象的
name 指向相同的地址        name = s.name;        //深拷贝
//name = new char(20);        //memcpy(name, s.name,
strlen(s.name));        cout << "copy Student" << endl;    };};
int main(){        {// 花括号让 s1 和 s2 变成局部对象，方便测试
Student s1;                Student s2(s1);// 复制对象        }
system("pause");        return 0;}//浅拷贝执行结果：//Student//copy
Student//~Student 0x7fffed0c3ec0//~Student 0x7fffed0c3ed0//***
Error in `/tmp/815453382/a.out': double free or corruption
(fasttop): 0x0000000001c82c20 ***//深拷贝执行结果：//Student//copy
Student//~Student 0x7fffebca9fb0//~Student 0x7fffebca9fc0
```

从执行结果可以看出，浅拷贝在对象的拷贝创建时存在风险，即被拷贝的对象析构释放资源之后，拷贝对象析构时会再次释放一个已经释放的资源，深拷贝的结果是两个对象之间没有任何关系，各自成员地址不同。

36、内联函数和宏定义的区别

- 在使用时，宏只做简单字符串替换（编译前）。而内联函数可以进行参数类型检查（编译时），且具有返回值。
- 内联函数在编译时直接将函数代码嵌入到目标代码中，省去函数调用的开销来提高执行效率，并且进行参数类型检查，具有返回值，可以实现重载。
- 宏定义时要注意书写（参数要括起来）否则容易出现歧义，内联函数不会产生歧义
- 内联函数有类型检测、语法判断等功能，而宏没有

感谢网友“bygaoyuan”重新整理，
https://github.com/forthespada/InterviewGuide/issues/3

内联函数适用场景:

- 使用宏定义的地方都可以使用 inline 函数。
- 作为类成员接口函数来读写类的私有成员或者保护成员，会提高效率。

37、public，protected 和 private 访问和继承权限 /public/protected/private 的区别?

- public 的变量和函数在类的内部外部都可以访问。
- protected 的变量和函数只能在类的内部和其派生类中访问。
- private 修饰的元素只能在类内访问。

（一）访问权限

派生类可以继承基类中除了构造/析构、赋值运算符重载函数之外的成员，但是这些成员的访问属性在派生过程中也是可以调整的，三种派生方式的访问权限如下表所示：注意外部访问并不是真正的外部访问，而是在通过派生类的对象对基类成员的访问。

派生类对基类成员的访问形象有如下两种：

- 内部访问：由派生类中新增的成员函数对从基类继承来的成员的访问
- **外部访问**：在派生类外部，通过派生类的对象对从基类继承来的成员的访问

（二）继承权限

public 继承

公有继承的特点是基类的公有成员和保护成员作为派生类的成员时，都保持原有的状态，而基类的私有成员任然是私有的，不能被这个派生类的子类所访问

protected 继承

保护继承的特点是基类的所有公有成员和保护成员都成为派生类的保护成员，并且只能被它的派生类成员函数或友元函数访问，基类的私有成员仍然是私有的，访问规则如下表

private 继承

私有继承的特点是基类的所有公有成员和保护成员都成为派生类的私有成员，并不被它的派生类的子类所访问，基类的成员只能由自己派生类访问，无法再往下继承，访问规则如下表

38、如何用代码判断大小端存储

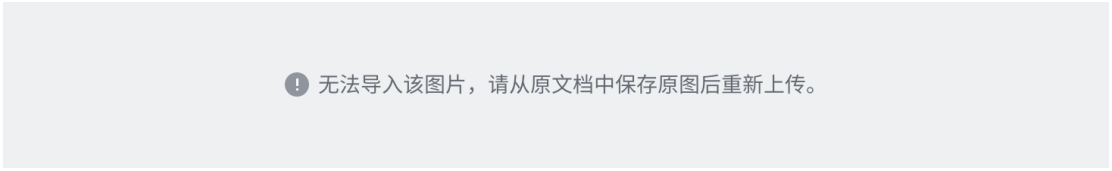
大端存储：字数据的高字节存储在低地址中

小端存储：字数据的低字节存储在低地址中

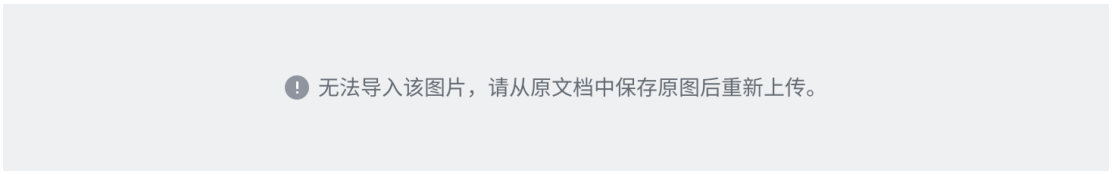
例如：32bit 的数字 0x12345678

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

小端模式中的存储方式为：



大端模式中的存储方式为：



了解了大小端存储的方式，如何在代码中进行判断呢？下面介绍两种判断方式：

方式一：使用强制类型转换-这种法子不错

```
Plaintext
#include <iostream>using namespace std;int main(){    int a = 0x1234;    //由于 int 和 char 的长度不同，借助 int 型转换成 char 型，只会留下低地址的部分    char c = (char)(a);    if (c == 0x12) cout << "big endian" << endl;    else if(c == 0x34)        cout << "little endian" << endl;}
```

方式二：巧用 union 联合体

```
Plaintext
#include <iostream>using namespace std;//union 联合体的重叠式存储，endian 联合体占用内存的空间为每个成员字节长度的最大值 union endian{    int a;    char ch;};int main(){    endian value; value.a = 0x1234;    //a 和 ch 共用 4 字节的内存空间    if (value.ch == 0x00)        cout << "big endian"<<endl;    else         cout << "little endian"<<endl;}
```

update1：感谢微信好友“李宇杰”指出方式二中代码错误，感谢。

39、volatile、mutable 和 explicit 关键字的用法

(1)**volatile**

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**，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

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。

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。

volatile 指针

volatile 指针和 const 修饰词类似，const 有常量指针和指针常量的说法，volatile 也有相应的概念

修饰由指针指向的对象、数据是 const 或 volatile 的：

```
Plaintext
const char* cpch;volatile char* vpch;
```

指针自身的值——一个代表地址的整数变量，是 const 或 volatile 的：

```
Plaintext
char* const pchc;char* volatile pchv;
```

注意：

- 可以把一个非 volatile int 赋给 volatile int，但是不能把非 volatile 对象赋给一个 volatile 对象。
- 除了基本类型外，对用户定义类型也可以用 volatile 类型进行修饰。
- C++中一个有 volatile 标识符的类只能访问它接口的子集，一个由类的实现者控制的子集。用户只能用 const_cast 来获得对类型接口的完全访问。此外，volatile 向 const 一样会从类传递到它的成员。

多线程下的 volatile

有些变量是用 volatile 关键字声明的。当两个线程都要用到某一个变量且该变量的值会被改变时，应该用 volatile 声明，**该关键字的作用是防止优化编译器把变量从内存装入 CPU 寄存器中。**如果变量被装入寄存器，那么两个线程有可能一个使用内存中的变量，一个使用寄存器中的变量，这会造成程序的错误执行。volatile 的意思是让编译器每次操作该变量时一定要从内存中真正取出，而不是使用已经存在寄存器中的值。

（2）**mutable**

mutable 的中文意思是“可变的，易变的”，跟 constant（既 C++中的 const）是反义词。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，即使在一个 const 函数中。我们知道，如果类的成员函数不会改变对象的状态，那么这个成员函数一般会声明成 const 的。但是，有些时候，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**。

样例

```
Plaintext
class person{int m_A;mutable int m_B;//特殊变量 在常函数里值也可以被修改 public:     void add() const//在函数里不可修改 this 指针指向的值常量指针     {         m_A=10;//错误  不可修改值，this 已经被修饰为常量指针         m_B=20;//正确     }}class person{int m_A;mutable int m_B;//特殊变量 在常函数里值也可以被修改}int main(){const person p;//修饰常对象 不可修改类成员的值 p.m_A=10;//错误，被修饰了指针常量 p.m_B=200;//正确，特殊变量，修饰了 mutable}
```

（3）**explicit**

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**，注意以下几点：

- explicit 关键字只能用于类内部的构造函数声明上
- explicit 关键字作用于单个参数的构造函数
- 被 explicit 修饰的构造函数的类，不能发生相应的隐式类型转换

update1:感谢网友“bygaoyuyan”给出修改意见，已采纳。
https://github.com/forthespada/InterviewGuide/issues/5

40、什么情况下会调用拷贝构造函数

- 用类的一个实例化对象去初始化另一个对象的时候
- 函数的参数是类的对象时（非引用传递）
- 函数的返回值是函数体内局部对象的类的对象时 ,此时虽然发生（Named return Value 优化）NRV 优化，但是由于返回方式是值传递，所以会在返回值的地方调用拷贝构造函数

另：第三种情况在 Linux g++ 下则不会发生拷贝构造函数，不仅如此即使返回局部对象的引用，依然不会发生拷贝构造函数

总结就是：即使发生 NRV 优化的情况下，Linux+ g++的环境是不管值返回方式还是

引用方式返回的方式都不会发生拷贝构造函数，而 Windows + VS2019 在值返回的情况下发生拷贝构造函数，引用返回方式则不发生拷贝构造函数。

在 c++编译器发生 NRV 优化，如果是引用返回的形式则不会调用拷贝构造函数，如果是值传递的方式依然会发生拷贝构造函数。

在 VS2019 下进行下述实验：

举个例子:

```
Plaintext
class A{public:        A() {};        A(const A& a)
{                cout << "copy constructor is called" <<
endl;        };        ~A() {};};void useClassA(A a) {}A
getClassA()//此时会发生拷贝构造函数的调用，虽然发生 NRV 优化，但是依然调用拷贝构造函数{        A a;        return a;}//A& getClassA2()//
VS2019 下，此时编辑器会进行（Named return Value 优化）NRV 优化,不调用拷贝构造函数 ，如果是引用传递的方式返回当前函数体内生成的对象时，并不发生拷贝构造函数的调用//{//        A a;//        return a;//}int
main(){        A a1, a2,a3,a4;        A a2 = a1;  //调用拷贝构造函数,对应情况 1        useClassA(a1);//调用拷贝构造函数，对应情况 2
a3 = getClassA();//发生 NRV 优化，但是值返回，依然会有拷贝构造函数的调用 情况 3        a4 = getClassA2(a1);//发生 NRV 优化，且引用返回自身，不会调用    return 0;}
```

情况 1 比较好理解

情况 2 的实现过程是，调用函数时先根据传入的实参产生临时对象，再用拷贝构造去初始化这个临时对象，在函数中与形参对应，函数调用结束后析构临时对象

情况 3 在执行 return 时，理论的执行过程是：产生临时对象，调用拷贝构造函数把返回对象拷贝给临时对象，函数执行完先析构局部变量，再析构临时对象， 依然会调用拷贝构造函数

update1:https://github.com/forthespada/InterviewGuide/issues/2 提出，感谢！-2021.03.22

41、C++中有几种类型的 new

在 C++中，new 有三种典型的使用方法：plain new，nothrow new 和 placement new

（1）**plain new**

言下之意就是普通的 new，就是我们常用的 new，在 C++中定义如下:

```
Plaintext
```

```
void* operator new(std::size_t) throw(std::bad_alloc);void
operator delete(void *) throw();
```

因此 **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，因此通过判断返回值是否为 NULL 是徒劳的，举个例子：

```
Plaintext
#include <iostream>#include <string>using namespace std;int
main(){        try        {                char *p = new
char[10e11];               delete p;        }        catch (const
std::bad_alloc &ex)        {                cout << ex.what() <<
endl;        }        return 0;}//执行结果：bad allocation
```

（2）**nothrow new**

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

```
Plaintext
void * operator new(std::size_t,const std::nothrow_t&)
throw();void operator delete(void*) throw();
```

举个例子：

```
Plaintext
#include <iostream>#include <string>using namespace std;int
main(){        char *p = new(nothrow) char[10e11];        if (p ==
NULL)        {                cout << "alloc failed" <<
endl;        }        delete p;        return 0;}//运行结果：alloc
failed
```

（3）**placement new**

这种 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。定义如下：

```
Plaintext
void* operator new(size_t,void*);void operator
delete(void*,void*);
```

使用 placement new 需要注意两点：

- 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。

举个例子：

42、C++的异常处理的方法

在程序执行过程中，由于程序员的疏忽或是系统资源紧张等因素都有可能导致异常，任何程序都无法保证绝对的稳定，常见的异常有：

- 数组下标越界
- 除法计算时除数为 0
- 动态分配空间时空间不足
- ...

如果不及时对这些异常进行处理，程序多数情况下都会崩溃。

（1）try、throw 和 catch 关键字

C++中的异常处理机制主要使用 **try**、**throw** 和 **catch** 三个关键字，其在程序中的用法如下：

```
Plaintext
#include <iostream>using namespace std;int main(){    double m = 1, n = 0;    try {        cout << "before dividing." << endl;        if (n == 0)            throw - 1;  //抛出 int 型异常        else if (m == 0)            throw - 1.0;  //抛出 double 型异常        else cout << m / n << endl;        cout << "after dividing." << endl;    }    catch (double d) {        cout << "catch (double)" << d << endl;    }    catch (...) {        cout << "catch (...)" << endl;    }    cout << "finished" << endl;    return 0;}//运行结果//before dividing.//catch (...)//finished
```

代码中，对两个数进行除法计算，其中除数为 0。可以看到以上三个关键字，程序的执行流程是先执行 try 包裹的语句块，如果执行过程中没有异常发生，则不会进入任何 catch 包裹的语句块，如果发生异常，则使用 throw 进行异常抛出，再由 catch 进行捕获，throw 可以抛出各种数据类型的信息，代码中使用的是数字，也可以自定义异常 class。**catch 根据 throw 抛出的数据类型进行精确捕获（不会出现类型转换），如果匹配不到就直接报错，可以使用 catch(...)的方式捕获任何异常（不推荐）。**当然，如果 catch 了异常，当前函数如果不进行处理，或者已经处理了想通知上一层的调用者，可以在 catch 里面再 throw 异常。

（2）函数的异常声明列表

有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：

```
Plaintext
int fun() throw(int,double,A,B,C){...};
```

这种写法表名函数可能会抛出 int,double 型或者 A、B、C 三种类型的异常，如果 throw 中为空，表明不会抛出任何异常，如果没有 throw 则可能抛出任何异常

（3）C++标准异常类 exception

C++ 标准库中有一些类代表异常，这些类都是从 exception 类派生而来的，如下图所示

- bad_typeid：使用 typeid 运算符，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会抛出此异常，例如：

```
Plaintext
#include <iostream>#include <typeinfo>using namespace std;class
A{public:  virtual ~A();}; using namespace std;int main()
{        A* a = NULL;        try {                  cout <<
typeid(*a).name() << endl; // Error condition          }
catch (bad_typeid){                  cout << "Object is NULL" <<
endl;          }    return 0;}//运行结果：bject is NULL
```

- bad_cast：在用 dynamic_cast 进行从多态基类对象（或引用）到派生类的引用的强制类型转换时，如果转换是不安全的，则会抛出此异常
- bad_alloc：在用 new 运算符进行动态内存分配时，如果没有足够的内存，则会引发此异常
- out_of_range:用 vector 或 string 的 at 成员函数根据下标访问元素时，如果下标越界，则会抛出此异常

43、static 的用法和作用?

1.先来介绍它的第一条也是最重要的一条：隐藏。（static 函数，static 变量均可）

当同时编译多个文件时，所有未加 static 前缀的全局变量和函数都具有全局可见性。

2.static 的第二个作用是保持变量内容的持久。（static 变量中的记忆功能和全局生存期）存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始

化。共有两种变量存储在静态存储区：全局变量和 static 变量，只不过和全局变量比起来，static 可以控制变量的可见范围，说到底 static 还是用来隐藏的。

3.static 的第三个作用是默认初始化为 0（static 变量）

其实全局变量也具备这一属性，因为全局变量也存储在静态数据区。在静态数据区，内存中所有的字节默认值都是 0x00，某些时候这一特点可以减少程序员的工作量。

4.static 的第四个作用：C++中的类成员声明 static

1. 函数体内 static 变量的作用范围为该函数体，不同于 auto 变量，该变量的内存只被分配一次，因此其值在下次调用时仍维持上次的值；

2. 在模块内的 static 全局变量可以被模块内所用函数访问，但不能被模块外其它函数访问；

3.

在模块内的 static 函数只可被这一模块内的其它函数调用，这个函数的使用范围被限制在声明它的模块内；

4. 在类中的 static 成员变量属于整个类所拥有，对类的所有对象只有一份拷贝；

5. 在类中的 static 成员函数属于整个类所拥有，这个函数不接收 this 指针，因而只能访问类的 static 成员变量。

类内：

1. static 类对象必须要在类外进行初始化，static 修饰的变量先于对象存在，所以 static 修饰的变量要在类外初始化；

2. 由于 static 修饰的类成员属于类，不属于对象，因此 static 类成员函数是没有 this 指针的，this 指针是指向本对象的指针。正因为没有 this 指针，所以 static 类成员函数不能访问非 static 的类成员，只能访问 static 修饰的类成员；

3. static 成员函数不能被 virtual 修饰，static 成员不属于任何对象或实例，所以加上 virtual 没有任何实际意义；静态成员函数没有 this 指针，虚函数的实现是为每一个对象分配一个 vptr 指针，而 vptr 是通过 this 指针调用的，所以不能为 virtual；虚函数的调用关系，this->vptr->ctable->virtual function

44、指针和 const 的用法

1. 当 const 修饰指针时，由于 const 的位置不同，它的修饰对象会有所不同。

2. int *const p2 中 const 修饰 p2 的值,所以理解为 p2 的值不可以改变，即 p2 只能指向固定的一个变量地址，但可以通过* p2 读写这个变量的值。顶层指针表示指针本身是一个常量

3. int const *p1 或者 const int p1 两种情况中 const 修饰 p1，所以理解为* p1 的值不可

以改变，即不可以给*p1 赋值改变 p1 指向变量的值，但可以通过给 p 赋值不同的地址改变这个指针指向。

底层指针表示指针所指向的变量是一个常量。

45、形参与实参的区别?

1. 形参变量只有在被调用时才分配内存单元，在调用结束时， 即刻释放所分配的内存单元。因此，形参只有在函数内部有效。 函数调用结束返回主调函数后则不能再使用该形参变量。

2. 实参可以是常量、变量、表达式、函数等， 无论实参是何种类型的量，在进行函数调用时，它们都必须具有确定的值， 以便把这些值传送给形参。 因此应预先用赋值，输入等办法使实参获得确定值，会产生一个临时变量。

3. 实参和形参在数量上，类型上，顺序上应严格一致， 否则会发生“类型不匹配”的错误。

4. 函数调用中发生的数据传送是单向的。 即只能把实参的值传送给形参，而不能把形参的值反向地传送给实参。 因此在函数调用过程中，形参的值发生改变，而实参中的值不会变化。

5. 当形参和实参不是指针类型时，在该函数运行时，形参和实参是不同的变量，他们在内存中位于不同的位置，形参将实参的内容复制一份，在该函数运行结束的时候形参被释放，而实参内容不会改变。

46、值传递、指针传递、引用传递的区别和效率

1. 值传递：有一个形参向函数所属的栈拷贝数据的过程，如果值传递的对象是类对象 或是大的结构体对象，将耗费一定的时间和空间。(传值)

2. 指针传递：同样有一个形参向函数所属的栈拷贝数据的过程，但拷贝的数据是一个固定为 4 字节的地址。(传值，传递的是地址值)

3. 引用传递：同样有上述的数据拷贝过程，但其是针对地址的，相当于为该数据所在的地址起了一个别名。(传地址)

4. 效率上讲，指针传递和引用传递比值传递效率高。一般主张使用引用传递，代码逻辑上更加紧凑、清晰。

47、静态变量什么时候初始化

1. 初始化只有一次，但是可以多次赋值，在主程序之前，编译器已经为其分配好了内存。

2. 静态局部变量和全局变量一样，数据都存放在全局区域，所以在主程序之前，编译器已经为其分配好了内存，但在 C 和 C++中静态局部变量的初始化节点又有点不太

一样。在 C 中，初始化发生在代码执行之前，编译阶段分配好内存之后，就会进行初始化，所以我们看到在 C 语言中无法使用变量对静态局部变量进行初始化，在程序运行结束，变量所处的全局内存会被全部回收。

3. 而在 C++中，初始化时在执行相关代码时才会进行初始化，主要是由于 C++引入对象后，要进行初始化必须执行相应构造函数和析构函数，在构造函数或析构函数中经常会需要进行某些程序中需要进行的特定操作，并非简单地分配内存。所以 C++标准定为全局或静态对象是有首次用到时才会进行构造，并通过 atexit()来管理。在程序结束，按照构造顺序反方向进行逐个析构。所以在 C++中是可以使用变量对静态局部变量进行初始化的。

48、const 关键字的作用有哪些?

1. 阻止一个变量被改变，可以使用 const 关键字。在定义该 const 变量时，通常需要对它进行初始化，因为以后就没有机会再去改变它了；

2. 对指针来说，可以指定指针本身为 const，也可以指定指针所指的数据为 const，或二者同时指定为 const；

3. 在一个函数声明中，const 可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值；

4. 对于类的成员函数，若指定其为 const 类型，则表明其是一个常函数，不能修改类的成员变量，类的常对象只能访问类的常成员函数；

5. 对于类的成员函数，有时候必须指定其返回值为 const 类型，以使得其返回值不为“左值”。

6. const 成员函数可以访问非 const 对象的非 const 数据成员、const 数据成员，也可以访问 const 对象内的所有数据成员；

7. 非 const 成员函数可以访问非 const 对象的非 const 数据成员、const 数据成员，但不可以访问 const 对象的任意数据成员；

8. 一个没有明确声明为 const 的成员函数被看作是将要修改对象中数据成员的函数，而且编译器不允许它为一个 const 对象所调用。因此 const 对象只能调用 const 成员函数。

9. const 类型变量可以通过类型转换符 const_cast 将 const 类型转换为非 const 类型；

10. const 类型变量必须定义的时候进行初始化，因此也导致如果类的成员变量有 const 类型的变量，那么该变量必须在类的初始化列表中进行初始化；

11. 对于函数值传递的情况，因为参数传递是通过复制实参创建一个临时变量传递进函数的，函数内只能改变临时变量，但无法改变实参。则这个时候无论加不加 const 对实参不会产生任何影响。但是在引用或指针传递函数调用中，因为传进去的是一个引用或指针，这样函数内部可以改变引用或指针所指向的变量，这时 const 才是实实

在在地保护了实参所指向的变量。因为在编译阶段编译器对调用函数的选择是根据实参进行的，所以，只有引用传递和指针传递可以用是否加 const 来重载。一个拥有顶层 const 的形参无法和另一个没有顶层 const 的形参区分开来。

49、什么是类的继承?

1. 类与类之间的关系

has-A 包含关系，用以描述一个类由多个部件类构成，实现 has-A 关系用类的成员属性表示，即一个类的成员属性是另一个已经定义好的类；

use-A，一个类使用另一个类，通过类之间的成员函数相互联系，定义友元或者通过传递参数的方式来实现；

is-A，继承关系，关系具有传递性；

2.继承的相关概念

所谓的继承就是一个类继承了另一个类的属性和方法，这个新的类包含了上一个类的属性和方法，被称为子类或者派生类，被继承的类称为父类或者基类；

3.继承的特点

子类拥有父类的所有属性和方法，子类可以拥有父类没有的属性和方法，子类对象可以当做父类对象使用；

4.继承中的访问控制

public、protected、private

5.继承中的构造和析构函数

6.继承中的兼容性原则

50、从汇编层去解释一下引用

```
Plaintext
9:      int x = 1;00401048  mov     dword ptr [ebp-4],110:     int &b = x;0040104F   lea     eax,[ebp-4]00401052  mov     dword ptr [ebp-8],eax
```

x 的地址为 ebp-4，b 的地址为 ebp-8，因为栈内的变量内存是从高往低进行分配的，所以 b 的地址比 x 的低。

lea eax,[ebp-4] 这条语句将 x 的地址 ebp-4 放入 eax 寄存器

mov dword ptr [ebp-8],eax 这条语句将 eax 的值放入 b 的地址

ebp-8 中上面两条汇编的作用即：将 x 的地址存入变量 b 中，这不和将某个变量的地址存入指针变量是一样的吗？所以从汇编层次来看，的确引用是通过指针来实现的。

51、深拷贝与浅拷可以描述一下吗?

浅复制： 只是拷贝了基本类型的数据，而引用类型数据，复制后也是会发生引用，我们把这种拷贝叫做“（浅复制）浅拷贝”，换句话说，浅复制仅仅是指向被复制的内存地址，如果原地址中对象被改变了，那么浅复制出来的对象也会相应改变。

深复制： 在计算机中开辟了一块新的内存地址用于存放复制的对象。

在某些状况下，类内成员变量需要动态开辟堆内存，如果实行位拷贝，也就是把对象里的值完全复制给另一个对象，如 A=B。这时，如果 B 中有一个成员变量指针已经申请了内存，那 A 中的那个成员变量也指向同一块内存。这就出现了问题：当 B 把内存释放了（如：析构），这时 A 内的指针就是野指针了，出现运行错误。

52、new 和 malloc 的区别?

1、 new/delete 是 C++关键字，需要编译器支持。malloc/free 是库函数，需要头文件支持;

2、 使用 new 操作符申请内存分配时无须指定内存块的大小，编译器会根据类型信息自行计算。而 malloc 则需要显式地指出所需内存的尺寸。

3、 new 操作符内存分配成功时，返回的是对象类型的指针，类型严格与对象匹配，无须进行类型转换，故 new 是符合类型安全性的操作符。而 malloc 内存分配成功则是返回 void * ，需要通过强制类型转换将 void*指针转换成我们需要的类型。

4、 new 内存分配失败时，会抛出 bac_alloc 异常。malloc 分配内存失败时返回 NULL。

5、 new 会先调用 operator new 函数，申请足够的内存（通常底层使用 malloc 实现）。然后调用类型的构造函数，初始化成员变量，最后返回自定义类型指针。delete 先调用析构函数，然后调用 operator delete 函数释放内存（通常底层使用 free 实现）。malloc/free 是库函数，只能动态的申请和释放内存，无法强制要求其做自定义类型对象构造和析构工作。

53、delete p、delete [] p、allocator 都有什么作用?

1、 动态数组管理 new 一个数组时，[]中必须是一个整数，但是不一定是常量整数，普通数组必须是一个常量整数;

2、 new 动态数组返回的并不是数组类型，而是一个元素类型的指针;

3、 delete[]时，数组中的元素按逆序的顺序进行销毁;

4、 new 在内存分配上面有一些局限性，new 的机制是将内存分配和对象构造组合在一起，同样的，delete 也是将对象析构和内存释放组合在一起的。allocator 将这两部分分开进行，allocator 申请一部分内存，不进行初始化对象，只有当需要的时候才进行初始化操作。

54、new 和 delete 的实现原理， delete 是如何知道释放内存的大小的额?

1、 new 简单类型直接调用 operator new 分配内存;

而对于复杂结构，先调用 operator new 分配内存，然后在分配的内存上调用构造函数;

对于简单类型，new[]计算好大小后调用 operator new;

对于复杂数据结构，new[]先调用 operator new[]分配内存，然后在 p 的前四个字节写入数组大小 n，然后调用 n 次构造函数，针对复杂类型，new[]会额外存储数组大小;

① new 表达式调用一个名为 operator new(operator new[])函数，分配一块足够大的、原始的、未命名的内存空间;

② 编译器运行相应的构造函数以构造这些对象，并为其传入初始值;

③ 对象被分配了空间并构造完成，返回一个指向该对象的指针。

2、 delete 简单数据类型默认只是调用 free 函数；复杂数据类型先调用析构函数再调用 operator delete；针对简单类型，delete 和 delete[]等同。假设指针 p 指向 new[]分配的内存。因为要 4 字节存储数组大小，实际分配的内存地址为[p-4]，系统记录的也是这个地址。delete[]实际释放的就是 p-4 指向的内存。而 delete 会直接释放 p 指向的内存，这个内存根本没有被系统记录，所以会崩溃。

3、 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。

55、malloc 申请的存储空间能用 delete 释放吗

不能，malloc /free 主要为了兼容 C，new 和 delete 完全可以取代 malloc /free 的。

malloc /free 的操作对象都是必须明确大小的，而且不能用在动态类上。

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。

当然从理论上说使用 malloc 申请的内存是可以通过 delete 释放的。不过一般不这样写的。而且也不能保证每个 C++的运行时都能正常。

56、malloc 与 free 的实现原理?

1、 在标准 C 库中，提供了 malloc/free 函数分配释放内存，这两个函数底层是由 brk、mmap、, munmap 这些系统调用实现的;

2、 brk 是将数据段(.data)的最高地址指针_edata 往高地址推,mmap 是在进程的虚拟

地址空间中（堆和栈中间，称为文件映射区域的地方）找一块空闲的虚拟内存。这两种方式分配的都是虚拟内存，没有分配物理内存。在第一次访问已分配的虚拟地址空间的时候，发生缺页中断，操作系统负责分配物理内存，然后建立虚拟内存和物理内存之间的映射关系；

3、 malloc 小于 128k 的内存，使用 brk 分配内存，将_edata 往高地址推；malloc 大于 128k 的内存，使用 mmap 分配内存，在堆和栈之间找一块空闲内存分配；brk 分配的内存需要等到高地址内存释放以后才能释放，而 mmap 分配的内存可以单独释放。当最高地址空间的空闲内存超过 128K（可由 M_TRIM_THRESHOLD 选项调节）时，执行内存紧缩操作（trim）。在上一个步骤 free 的时候，发现最高地址空闲内存超过 128K，于是内存紧缩。

4、 malloc 是从堆里面申请内存，也就是说函数返回的指针是指向堆里面的一块内存。操作系统中有一个记录空闲内存地址的链表。当操作系统收到程序的申请时，就会遍历该链表，然后就寻找第一个空间大于所申请空间的堆结点，然后就将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。

57、malloc、realloc、calloc 的区别

1. malloc 函数

```
Plaintext
void* malloc(unsigned int num_size);int *p =
malloc(20*sizeof(int));申请 20 个 int 类型的空间；
```

1. calloc 函数

```
Plaintext
void* calloc(size_t n,size_t size);int *p = calloc(20,
sizeof(int));
```

省去了人为空间计算；malloc 申请的空间的值是随机初始化的，calloc 申请的空间的值是初始化为 0 的；

1. realloc 函数

```
Plaintext
void realloc(void *p, size_t new_size);
```

给动态分配的空间分配额外的空间，用于扩充容量。

58、类成员初始化方式？构造函数的执行顺序？为什么用成员初始化列表会快一些？

1. 赋值初始化，通过在函数体内进行赋值初始化；列表初始化，在冒号后使用初始化列表进行初始化。

这两种方式的主要区别在于：

对于在函数体中初始化,是在所有的数据成员被分配内存空间后才进行的。

列表初始化是给数据成员分配内存空间时就进行初始化,就是说分配一个数据成员只要冒号后有此数据成员的赋值表达式(此表达式必须是括号赋值表达式),那么分配了内存空间后在进入函数体之前给数据成员赋值，就是说初始化这个数据成员此时函数体还未执行。

1. 一个派生类构造函数的执行顺序如下：

① 虚拟基类的构造函数（多个虚拟基类则按照继承的顺序执行构造函数）。

② 基类的构造函数（多个普通基类也按照继承的顺序执行构造函数）。

③ 类类型的成员对象的构造函数（按照初始化顺序）

④ 派生类自己的构造函数。

1. 方法一是在构造函数当中做赋值的操作，而方法二是做纯粹的初始化操作。我们都知道，C++的赋值操作是会产生临时对象的。临时对象的出现会降低程序的效率。

59、有哪些情况必须用到成员列表初始化？作用是什么？

1. 必须使用成员初始化的四种情况

① 当初始化一个引用成员时；

② 当初始化一个常量成员时；

③ 当调用一个基类的构造函数，而它拥有一组参数时；

④ 当调用一个成员类的构造函数，而它拥有一组参数时；

1. 成员初始化列表做了什么

① 编译器会一一操作初始化列表，以适当的顺序在构造函数之内安插初始化操作，并且在任何显示用户代码之前；

② list 中的项目顺序是由类中的成员声明顺序决定的，不是由初始化列表的顺序决定的；

参考资料：《C++对象模型》P74

update1:感谢网友“lcf163”提出修改意见，已采纳。
https://github.com/forthespada/InterviewGuide/issues/4

60、C++中新增了 string，它与 C 语言中的 char *有什么区别吗？它是如何实现的？

string 继承自 basic_string,其实是对 char *进行了封装，封装的 string 包含了 char* 数组，容量，长度等等属性。

string 可以进行动态扩展，在每次扩展的时候另外申请一块原空间大小两倍的空间（2^n），然后将原字符串拷贝过去，并加上新增的内容。

61、什么是内存泄露，如何检测与避免

内存泄露

一般我们常说的内存泄漏是指**堆内存的泄漏**。堆内存是指程序从堆中分配的，大小任意的(内存块的大小可以在程序运行期决定)内存块，使用完后必须显式释放的内存。应用程序般使用 malloc,、realloc、 new 等函数从堆中分配到块内存，使用完后，程序必须负责相应的调用 free 或 delete 释放该内存块，否则，这块内存就不能被再次使用，我们就说这块内存泄漏了

避免内存泄露的几种方式

- 计数法：使用 new 或者 malloc 时，让该数+1，delete 或 free 时，该数-1，程序执行完打印这个计数，如果不为 0 则表示存在内存泄露
- 一定要将基类的析构函数声明为**虚函数**
- 对象数组的释放一定要用 **delete []**
- 有 new 就有 delete，有 malloc 就有 free，保证它们一定成对出现

检测工具

- Linux 下可以使用 **Valgrind 工具**
- Windows 下可以使用 **CRT 库**

62、对象复用的了解，零拷贝的了解

对象复用

对象复用其本质是一种设计模式：Flyweight 享元模式。

通过将对象存储到“对象池”中实现对象的重复利用，这样可以避免多次创建重复对象的开销，节约系统资源。

零拷贝

零拷贝就是一种避免 CPU 将数据从一块存储拷贝到另外一块存储的技术。

零拷贝技术可以减少数据拷贝和共享总线操作的次数。

在 C++中，vector 的一个成员函数**emplace_back()**很好地体现了零拷贝技术，它跟 push_back()函数一样可以将一个元素插入容器尾部，区别在于：**使用 push_back()函数需要调用拷贝构造函数和转移构造函数，而使用 emplace_back()插入的元素原地构造，不需要触发拷贝构造和转移构造，**效率更高。举个例子：

```
Plaintext
#include <vector>#include <string>#include <iostream>using namespace std;struct Person{    string name;    int age;    //初始构造函数    Person(string p_name, int p_age): name(std::move(p_name)), age(p_age)    {         cout << "I have been constructed" <<endl;    }    //拷贝构造函数    Person(const Person& other): name(std::move(other.name)), age(other.age) {         cout << "I have been copy constructed" <<endl;    } //转移构造函数    Person(Person&& other): name(std::move(other.name)), age(other.age)    {         cout << "I have been moved"<<endl;    }};int main(){    vector<Person> e; cout << "emplace_back:" <<endl;    e.emplace_back("Jane", 23); //不用构造类对象    vector<Person> p;    cout << "push_back:"<<endl; p.push_back(Person("Mike",36));    return 0;}//输出结果：//emplace_back://I have been constructed//push_back://I have been constructed//I am being moved.
```

63、介绍面向对象的三大特性，并且举例说明

三大特性：继承、封装和多态

（1）继承

让某种类型对象获得另一个类型对象的属性和方法。

它可以使用现有类的所有功能，并在无需重新编写原来的类的情况下对这些功能进行扩展

常见的继承有三种方式：

1. 实现继承：指使用基类的属性和方法而无需额外编码的能力
2. 接口继承：指仅使用属性和方法的名称、但是子类必须提供实现的能力
3. 可视继承：指子窗体（类）使用基窗体（类）的外观和实现代码的能力（C++里好像不怎么用）

例如，将人定义为一个抽象类，拥有姓名、性别、年龄等公共属性，吃饭、睡觉、走路等公共方法，在定义一个具体的人时，就可以继承这个抽象类，既保留了公共属性和方法，也可以在此基础上扩展跳舞、唱歌等特有方法

（2）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。

封装，也就是**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏，例如：将公共的数据或方法使用 public 修饰，而不希望被访问的数据或方法采用 private 修饰。

（3）多态

同一事物表现出不同事物的能力，即向不同对象发送同一消息，不同的对象在接收时会产生不同的行为**（重载实现编译时多态，虚函数实现运行时多态）**。

多态性是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。**简单一句话：允许将子类类型的指针赋值给父类类型的指针**

实现多态有二种方式：覆盖（override），重载（overload）。

覆盖：是指子类重新定义父类的虚函数的做法。

重载：是指允许存在多个同名函数，而这些函数的参数表不同（或许参数个数不同，或许参数类型不同，或许两者都不同）。例如：基类是一个抽象对象——人，那教师、运动员也是人，而使用这个抽象对象既可以表示教师、也可以表示运动员。

64、成员初始化列表的概念，为什么用它会快一些?

成员初始化列表的概念

在类的构造函数中，不在函数体内对成员变量赋值，而是在构造函数的花括号前面使用冒号和初始化列表赋值

效率

用初始化列表会快一些的原因是，对于类型，它少了一次调用构造函数的过程，而在函数体中赋值则会多一次调用。而对于内置数据类型则没有差别。举个例子:

```
Plaintext
#include <iostream>using namespace std;class A{public:    A()
{        cout << "默认构造函数 A()" << endl;    }    A(int a)
{        value = a;        cout << "A(int "<<value<<")" <<
endl;    }    A(const A& a)    {        value = a.value;
cout << "拷贝构造函数 A(A& a):  "<<value << endl;    }    int
value;};class B{public:    B() : a(1)    {        b = A(2);    }
A a;    A b;};int main(){    B b;}//输出结果：//A(int 1)//默认构造函数 A()//A(int 2)
```

从代码运行结果可以看出，在构造函数体内部初始化的对象 b 多了一次构造函数的调用过程，而对象 a 则没有。由于对象成员变量的初始化动作发生在进入构造函数之前，

对于内置类型没什么影响，但**如果有些成员是类**，那么在进入构造函数之前，会先调用一次默认构造函数，进入构造函数后所做的事其实是一次赋值操作(对象已存在)，所以**如果是在构造函数体内进行赋值的话，等于是一次默认构造加一次赋值，而初始化列表只做一次赋值操作。**

65、C++的四种强制转换 reinterpret_cast/const_cast/static_cast /dynamic_cast

reinterpret_cast

reinterpret_cast (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。用法如下:

- 常量指针被转化成非常量的指针，并且仍然指向原来的对象
- 常量引用被转换成非常量的引用，并且仍然指向原来的对象
- const_cast 一般用于修改底指针。如 const char *p 形式

static_cast

static_cast < type-id > (expression)

该运算符把 expression 转换为 type-id 类型，但没有运行时类型检查来保证转换的安全性。它主要有如下几种用法:

- 用于类层次结构中基类（父类）和派生类（子类）之间指针或引用引用的转换
  - 进行上行转换（把派生类的指针或引用转换成基类表示）是安全的
  - 进行下行转换（把基类指针或引用转换成派生类表示）时，由于没有动态类型检查，所以是不安全的
- 用于基本数据类型之间的转换，如把 int 转换成 char，把 int 转换成 enum。这种转换的安全性也要开发人员来保证。
- 把空指针转换成目标类型的空指针
- 把任何类型的表达式转换成 void 类型

注意：static_cast 不能转换掉 expression 的 const、volatile、或者__unaligned 属性。

dynamic_cast

有类型检查，基类向派生类转换比较安全，但是派生类向基类转换则不太安全

dynamic_cast (expression)

该运算符把 expression 转换成 type-id 类型的对象。type-id 必须是类的指针、类的引用或者 void*

如果 type-id 是类指针类型，那么 expression 也必须是一个指针，如果 type-id 是一个引用，那么 expression 也必须是一个引用

dynamic_cast 运算符可以在执行期决定真正的类型，也就是说 expression 必须是多态类型。如果下行转换是安全的（也就说，如果基类指针或者引用确实指向一个派生类对象）这个运算符会传回适当转型过的指针。如果 如果下行转换不安全，这个运算符会传回空指针（也就是说，基类指针或者引用没有指向一个派生类对象）

dynamic_cast 主要用于类层次间的上行转换和下行转换，还可以用于类之间的交叉转换

在类层次间进行上行转换时，dynamic_cast 和 static_cast 的效果是一样的

在进行下行转换时，dynamic_cast 具有类型检查的功能，比 static_cast 更安全

举个例子:

```
Plaintext
#include <bits/stdc++.h>using namespace std;class Base{public:
Base() :b(1) {}        virtual void fun() {};        int b;};class
Son : public Base{public:        Son() :d(2) {}        int d;};int
main(){        int n = 97;        //reinterpret_cast        int *p
= &n;        //以下两者效果相同        char *c =
reinterpret_cast<char*> (p);        char *c2 =  (char*)(p);
cout << "reinterpret_cast 输出："<< *c2 << endl;
//const_cast        const int *p2 = &n;        int *p3 =
const_cast<int*>(p2);        *p3 = 100;        cout << "const_cast
输出：" << *p3 << endl;                Base* b1 = new Son;
Base* b2 = new Base;        //static_cast        Son* s1 =
static_cast<Son*>(b1); //同类型转换        Son* s2 =
static_cast<Son*>(b2); //下行转换，不安全        cout <<
"static_cast 输出："<< endl;        cout << s1->d << endl;
cout << s2->d << endl; //下行转换，原先父对象没有 d 成员，输出垃圾值
//dynamic_cast        Son* s3 = dynamic_cast<Son*>(b1); //同类型转
换        Son* s4 = dynamic_cast<Son*>(b2); //下行转换，安全
cout << "dynamic_cast 输出：" << endl;        cout << s3->d <<
endl;        if(s4 == nullptr)                cout << "s4 指针为
nullptr" << endl;        else                cout << s4->d <<
endl;                        return 0;}//输出结果//reinterpret_cast
```

```
输出：a//const_cast 输出：100//static_cast 输出：//2//-33686019//dynamic_cast 输出：//2//s4 指针为 nullptr
```

从输出结果可以看出，在进行下行转换时，dynamic_cast 安全的，如果下行转换不安全的话其会返回空指针，这样在进行操作的时候可以预先判断。而使用 static_cast 下行转换存在不安全的情况也可以转换成功，但是直接使用转换后的对象进行操作容易造成错误。

66、C++函数调用的压栈过程

以例子进行讲解

从代码入手，解释这个过程：

```
Plaintext
#include <iostream>using namespace std;int f(int n) {        cout << n << endl;        return n;}void func(int param1, int param2){        int var1 = param1;        int var2 = param2;        printf("var1=%d,var2=%d", f(var1), f(var2));//如果将 printf 换为 cout 进行输出，输出结果则刚好相反}int main(int argc, char* argv[]){        func(1, 2);        return 0;}//输出结果 //2//1//var1=1,var2=2
```

当函数从入口函数 main 函数开始执行时，编译器会将我们操作系统的运行状态，main 函数的返回地址、main 的参数、mian 函数中的变量、进行依次压栈;

当 main 函数开始调用 func()函数时，编译器此时会将 main 函数的运行状态进行压栈，再将 func()函数的返回地址、func()函数的参数从右到左、func()定义变量依次压栈;

当 func()调用 f()的时候，编译器此时会将 func()函数的运行状态进行压栈，再将的返回地址、f()函数的参数从右到左、f()定义变量依次压栈

从代码的输出结果可以看出，函数 f(var1)、f(var2)依次入栈，而后先执行 f(var2)，再执行 f(var1)，最后打印整个字符串，将栈中的变量依次弹出，最后主函数返回。

文字化表述

函数的调用过程：

1）从栈空间分配存储空间

2）从实参的存储空间复制值到形参栈空间

3）进行运算

形参在函数未调用之前都是没有分配存储空间的，在函数调用结束之后，形参弹出栈空间，清除形参空间。

数组作为参数的函数调用方式是地址传递，形参和实参都指向相同的内存空间，调用完成后，形参指针被销毁，但是所指向的内存空间依然存在，不能也不会被销毁。

当函数有多个返回值的时候，不能用普通的 return 的方式实现，需要通过传回地址的形式进行，即地址/指针传递。

67、写 C++代码时有一类错误是 coredump，很常见，你遇到过吗？怎么调试这个错误？

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。

- 使用 gdb 命令对 core 文件进行调试

以下例子在 Linux 上编写一段代码并导致 segment fault 并产生 core 文件

```
Plaintext
mkdir coredumpTestvim coredumpTest.cpp
```

在编辑器内键入

```
Plaintext
#include<stdio.h>int main(){    int i;    scanf("%d",i);//正确的应该是&i,这里使用 i 会导致 segment fault    printf("%d\n",i);    return 0;}
```

编译

```
Plaintext
g++ coredumpTest.cpp -g -o coredumpTest
```

运行

```
Plaintext
./coredumpTest
```

使用 gdb 调试 coredump

```
Plaintext
gdb [可执行文件名] [core 文件名]
```

68、说说移动构造函数

1. 我们用对象 a 初始化对象 b，后对象 a 我们就不在使用了，但是对象 a 的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把 a 对象的内容复制一份到 b 中，那么为什么我们不能直接使用 a 的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷；

2. 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制。浅层复制之所以危险，是因为两个指针共同指向一片内存空间，若第一个指针将其释放，另一个指针的指向就不合法了。

所以我们只要避免第一个指针释放空间就可以了。避免的方法就是将第一个指针（比如 a->value）置为 NULL，这样在调用析构函数的时候，由于有判断是否为 NULL 的语句，所以析构 a 的时候并不会回收 a->value 指向的空间；

1. 移动构造函数的参数和拷贝构造函数不同，拷贝构造函数的参数是一个左值引用，但是移动构造函数的初值是一个右值引用。意味着，移动构造函数的参数是一个右值或者将亡值的引用。也就是说，只用用一个右值，或者将亡值初始化另一个对象的时候，才会调用移动构造函数。而那个 move 语句，就是将一个左值变成一个将亡值。

69、C++中将临时变量作为返回值时的处理过程

首先需要明白一件事情，临时变量，在函数调用过程中是被压到程序进程的栈中的，当函数退出时，临时变量出栈，即临时变量已经被销毁，临时变量占用的内存空间没有被清空，但是可以被分配给其他变量，所以有可能在函数退出时，该内存已经被修改了，对于临时变量来说已经是没有意义的值了

C 语言里规定：16bit 程序中，返回值保存在 ax 寄存器中，32bit 程序中，返回值保持在 eax 寄存器中，如果是 64bit 返回值，edx 寄存器保存高 32bit，eax 寄存器保存低 32bit

由此可见，函数调用结束后，返回值被临时存储到寄存器中，并没有放到堆或栈中，也就是说与内存没有关系了。当退出函数的时候，临时变量可能被销毁，但是返回值却被放到寄存器中与临时变量的生命周期没有关系

如果我们需要返回值，一般使用赋值语句就可以了。

70、如何获得结构成员相对于结构开头的字节偏移量

使用<stddef.h>头文件中的，offsetof 宏。

举个例子：

```
Plaintext
#include <iostream>#include <stddef.h>using namespace std;struct
S{        int x;        char y;        int z;        double
a;};int main(){        cout << offsetof(S, x) << endl; // 0
```

```
cout << offsetof(S, y) << endl; // 4        cout << offsetof(S, z)
<< endl; // 8        cout << offsetof(S, a) << endl; // 12
return 0;}在 VS2019 + win 下 并不是这样的        cout << offsetof(S,
x) << endl; // 0        cout << offsetof(S, y) << endl; // 4
cout << offsetof(S, z) << endl; // 8        cout << offsetof(S, a)
<< endl; // 16 这里是 16 的位置，因为 double 是 8 字节，需要找一个 8 的倍
数对齐，当然了，如果加上 #pragma pack(4)指定 4 字节对齐就可以了
#pragma pack(4)struct S{        int x;        char y;        int
z;        double a;};void test02(){        cout << offsetof(S, x)
<< endl; // 0        cout << offsetof(S, y) << endl; // 4
cout << offsetof(S, z) << endl; // 8        cout << offsetof(S, a)
<< endl; // 12    }
```

S 结构体中各个数据成员的内存空间划分如下所示，需要注意内存对齐

71、静态类型和动态类型，静态绑定和动态绑定的介绍

- 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；

从上面的定义也可以看出，非虚函数一般都是静态绑定，而虚函数都是动态绑定（如此才可实现多态性）。 举个例子：

```Plaintext
#include <iostream>using namespace std;class A{public:
/*virtual*/ void func() { std::cout << "A::func()\n"; }};class B :
public A{public:        void func() { std::cout <<
"B::func()\n"; }};class C : public A{public:        void func()
{ std::cout << "C::func()\n"; }};int main(){        C* pc = new
C(); //pc 的静态类型是它声明的类型 C*，动态类型也是 C*;        B* pb =
new B(); //pb 的静态类型和动态类型也都是 B*;        A* pa = pc;
//pa 的静态类型是它声明的类型 A*，动态类型是 pa 所指向的对象 pc 的类型 C*;
pa = pb;        //pa 的动态类型可以更改，现在它的动态类型是 B*，但其静
```

```
态类型仍是声明时候的 A*；          C *pnull = NULL; //pnull 的静态类型是它声明的类型 C*,没有动态类型，因为它指向了 NULL；          pa->func(); //A::func() pa 的静态类型永远都是 A*，不管其指向的是哪个子类，都是直接调用 A::func()；          pc->func();       //C::func() pc 的动、静态类型都是 C*，因此调用 C::func()；          pnull->func();    //C::func() 不用奇怪为什么空指针也可以调用函数，因为这在编译期就确定了，和指针空不空没关系；          return 0;}
```

如果将 A 类中的 virtual 注释去掉，则运行结果是:

```
Plaintext
pa->func();       //B::func() 因为有了 virtual 虚函数特性，pa 的动态类型指向 B*，因此先在 B 中查找，找到后直接调用；pc->func(); //C::func() pc 的动、静态类型都是 C*，因此也是先在 C 中查找；pnull->func();    //空指针异常，因为是 func 是 virtual 函数，因此对 func 的调用只能等到运行期才能确定，然后才发现 pnull 是空指针；
```

在上面的例子中，

- 如果基类 A 中的 func 不是 virtual 函数，那么不论 pa、pb、pc 指向哪个子类对象，对 func 的调用都是在定义 pa、pb、pc 时的静态类型决定，早已在编译期确定了。
- 同样的空指针也能够直接调用 no-virtual 函数而不报错（这也说明一定要做空指针检查啊!），因此静态绑定不能实现多态;
- 如果 func 是虚函数，那所有的调用都要等到运行时根据其指向对象的类型才能确定，比起静态绑定自然是要有性能损失的，但是却能实现多态特性;

本文代码里都是针对指针的情况来分析的，但是对于引用的情况同样适用。

至此总结一下静态绑定和动态绑定的区别:

- 静态绑定发生在编译期，动态绑定发生在运行期;
- 对象的动态类型可以更改，但是静态类型无法更改;
- 要想实现动态，必须使用动态绑定;
- 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;

建议：

绝对不要重新定义继承而来的非虚(non-virtual)函数（《Effective C++ 第三版》条款 36），因为这样导致函数调用由对象声明时的静态类型确定了，而和对象本身脱离了关系，没有多态，也这将给程序留下不可预知的隐患和莫名其妙的 BUG；另外，在动态绑定也即在 virtual 函数中，要注意默认参数的使用。当缺省参数和 virtual 函数一起使用的时候一定要谨慎，不然出了问题怕是很难排查。 看下面的代码:

```
Plaintext
#include <iostream>using namespace std;class E{public:
virtual void func(int i = 0)        {                std::cout <<
"E::func()\t" << i << "\n";        }};class F : public E{public:
virtual void func(int i = 1)        {                std::cout <<
"F::func()\t" << i << "\n";        }};void test2(){        F* pf =
new F();        E* pe = pf;        pf->func(); //F::func() 1  正
常，就该如此；          pe->func(); //F::func() 0  哇哦，这是什么情况，
调用了子类的函数，却使用了基类中参数的默认值！}int
main(){        test2();        return 0;}
```

72、引用是否能实现动态绑定，为什么可以实现?

可以。

引用在创建的时候必须初始化，在访问虚函数时，编译器会根据其所绑定的对象类型决定要调用哪个函数。注意只能调用虚函数。

举个例子:

```
Plaintext
#include <iostream>using namespace std;class Base {public:
virtual void  fun()        {                cout << "base ::
fun()" << endl;        }};class Son : public Base{public:
virtual void  fun()        {                cout << "son :: fun()"
<< endl;        }        void func()        {                cout
<< "son :: not virtual function" <<endl;        }};int
main(){        Son s;        Base& b = s; // 基类类型引用绑定已经存
在的 Son 对象，引用必须初始化        s.fun(); //son::fun()
b.fun(); //son :: fun()        return 0;}
```

需要说明的是虚函数才具有动态绑定，上面代码中，Son 类中还有一个非虚函数 func()，这在 b 对象中是无法调用的，如果使用基类指针来指向子类也是一样的。

73、全局变量和局部变量有什么区别?

生命周期不同：全局变量随主程序创建和创建，随主程序销毁而销毁；局部变量在局部函数内部，甚至局部循环体等内部存在，退出就不存在；

使用方式不同：通过声明后全局变量在程序的各个部分都可以用到；局部变量分配在堆栈区，只能在局部使用。

操作系统和编译器通过内存分配的位置可以区分两者，全局变量分配在全局数据段并且在程序开始运行的时候被加载。局部变量则分配在堆栈里面 。

74、指针加减计算要注意什么?

指针加减本质是对其所指地址的移动，移动的步长跟指针的类型是有关系的，因此在涉及到指针加减运算需要十分小心，加多或者减多都会导致指针指向一块未知的内存地址，如果再进行操作就会很危险。

举个例子:

```
Plaintext
#include <iostream>using namespace std;int main(){        int *a, *b, c;        a = (int*)0x500;        b = (int*)0x520;        c = b - a;        printf("%d\n", c); // 8        a += 0x020;        c = b - a;        printf("%d\n", c); // -24        return 0;}
```

首先变量 a 和 b 都是以 16 进制的形式初始化，将它们转成 10 进制分别是 1280（5*16^2=1280）和 1312（516^2+2*16=1312)， 那么它们的差值为 32，也就是说 a 和 b 所指向的地址之间间隔 32 个位，但是考虑到是 int 类型占 4 位，所以 c 的值为 32/4=8

a 自增 16 进制 0x20 之后，其实际地址变为 1280 + 2*16*4 = 1408，（因为一个 int 占 4 位，所以要乘 4），这样它们的差值就变成了 1312 - 1280 = -96，所以 c 的值就变成了 -96/4 = -24

遇到指针的计算，**需要明确的是指针每移动一位，它实际跨越的内存间隔是指针类型的长度，建议都转成 10 进制计算，计算结果除以类型长度取得结果**

75、 怎样判断两个浮点数是否相等?

对两个浮点数判断大小和是否相等不能直接用==来判断，会出错！明明相等的两个数比较反而是不相等！对于两个浮点数比较只能通过相减并与预先设定的精度比较，记得要取绝对值！浮点数与 0 的比较也应该注意。与浮点数的表示方式有关。

76、方法调用的原理（栈，汇编）

1. 机器用栈来传递过程参数、存储返回信息、保存寄存器用于以后恢复，以及本地存储。而为单个过程分配的那部分栈称为帧栈；帧栈可以认为是程序栈的一段，它有两个端点，一个标识起始地址，一个标识着结束地址，两个指针结束地址指针 esp，开始地址指针 ebp;

2. 由一系列栈帧构成，这些栈帧对应一个过程，而且每一个栈指针+4 的位置存储函数返回地址；每一个栈帧都建立在调用者的下方，当被调用者执行完毕时，这一段栈帧会被释放。由于栈帧是向地址递减的方向延伸，因此如果我们将栈指针减去一定的值，就相当于给栈帧分配了一定空间的内存。如果将栈指针加上一定的值，也就是向上移动，那么就相当于压缩了栈帧的长度，也就是说内存被释放了。

3. 过程实现

① 备份原来的帧指针，调整当前的栈帧指针到栈指针位置；

② 建立起来的栈帧就是为被调用者准备的，当被调用者使用栈帧时，需要给临时变量分配预留内存；

③ 使用建立好的栈帧，比如读取和写入，一般使用 mov，push 以及 pop 指令等等。

④ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了

⑤ 恢复被调用者寄存器当中的值，这一过程其实是从栈帧中将备份的值再恢复到寄存器，不过此时这些值可能已经不在栈顶了。

⑥ 释放被调用者的栈帧，释放就意味着将栈指针加大，而具体的做法一般是直接将栈指针指向帧指针，因此会采用类似下面的汇编代码处理。

⑦ 恢复调用者的栈帧，恢复其实就是调整栈帧两端，使得当前栈帧的区域又回到了原始的位置。

⑧ 弹出返回地址，跳出当前过程，继续执行调用者的代码。

1. 过程调用和返回指令

① call 指令

② leave 指令

③ ret 指令

77、C++中的指针参数传递和引用参数传递有什么区别？底层原理你知道吗？

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。

被调函数对形参（本体）的任何操作都被处理成间接寻址，即通过栈中存放的地址访问主调函数中的实参变量（根据别名找到主调函数中的本体）。

因此，被调函数对形参的任何操作都会影响主调函数中的实参变量。

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。

78、类如何实现只能静态分配和只能动态分配

1. 前者是把 new、delete 运算符重载为 private 属性。后者是把构造、析构函数设为 protected 属性，再用子类来动态创建

2. 建立类的对象有两种方式：

① 静态建立，静态建立一个类对象，就是由编译器为对象在栈空间中分配内存；

② 动态建立，A *p = new A();动态建立一个类对象，就是使用 new 运算符为对象在堆空间中分配内存。这个过程分为两步，第一步执行 operator new()函数，在堆中搜索一块内存并进行分配；第二步调用类构造函数构造对象；

1. 只有使用 new 运算符，对象才会被建立在堆上，因此只要限制 new 运算符就可以实现类对象只能建立在栈上，可以将 new 运算符设为私有。

79、如果想将某个类用作基类，为什么该类必须定义而非声明?

派生类中包含并且可以使用它从基类继承而来的成员，为了使用这些成员，派生类必须知道他们是什么。

80、 继承机制中对象之间如何转换？指针和引用之间如何转换？

```
Plaintext
向上类型转换
```

将派生类指针或引用转换为基类的指针或引用被称为向上类型转换，向上类型转换会

自动进行，而且向上类型转换是安全的。

```
Plaintext
向下类型转换
```

将基类指针或引用转换为派生类指针或引用被称为向下类型转换，向下类型转换不会自动进行，因为一个基类对应几个派生类，所以向下类型转换时不知道对应哪个派生类，所以在向下类型转换时必须加动态类型识别技术。RTTI 技术，用 dynamic_cast 进行向下类型转换。

81、知道 C++中的组合吗？它与继承相比有什么优缺点吗？

一：继承

继承是 Is a 的关系，比如说 Student 继承 Person,则说明 Student is a Person。继承的优点是子类可以重写父类的方法来方便地实现对父类的扩展。

继承的缺点有以下几点：

①：父类的内部细节对子类是可见的。

②：子类从父类继承的方法在编译时就确定下来了，所以无法在运行期间改变从父类继承的方法的行为。

③：如果对父类的方法做了修改的话（比如增加了一个参数），则子类的方法必须做出相应的修改。所以说子类与父类是一种高耦合，违背了面向对象思想。

二：组合

组合也就是设计类的时候把要组合的类的对象加入到该类中作为自己的成员变量。

组合的优点：

①：当前对象只能通过所包含的那个对象去调用其方法，所以所包含的对象的内部细节对当前对象时不可见的。

②：当前对象与包含的对象是一个低耦合关系，如果修改包含对象的类中代码不需要修改当前对象类的代码。

③：当前对象可以在运行时动态的绑定所包含的对象。可以通过 set 方法给所包含对象赋值。

组合的缺点：①：容易产生过多的对象。②：为了能组合多个对象，必须仔细对接口进行定义。

82、函数指针？

1) 什么是函数指针?

函数指针指向的是特殊的数据类型，函数的类型是由其返回的数据类型和其参数列表共同决定的，而函数的名称则不是其类型的一部分。

一个具体函数的名字，如果后面不跟调用符号(即括号)，则该名字就是该函数的指针(注意：大部分情况下，可以这么认为，但这种说法并不很严格)。

2) 函数指针的声明方法

int (*pf)(const int&, const int&); (1)

上面的 pf 就是一个函数指针，指向所有返回类型为 int，并带有两个 const int&参数的函数。注意*pf 两边的括号是必须的，否则上面的定义就变成了：

int *pf(const int&, const int&); (2)

而这声明了一个函数 pf，其返回类型为 int *， 带有两个 const int&参数。

3) 为什么有函数指针

函数与数据项相似，函数也有地址。我们希望在同一个函数中通过使用相同的形参在不同的时间使用产生不同的效果。

4) 一个函数名就是一个指针，它指向函数的代码。

一个函数地址是该函数的进入点，也就是调用函数的地址。函数的调用可以通过函数名，也可以通过指向函数的指针来调用。函数指针还允许将函数作为变元传递给其他函数;

5) 两种方法赋值：

指针名 = 函数名； 指针名 = &函数名

83、说一说你理解的内存对齐以及原因

1.数据类型自身的对齐值：
对于 char 型数据，其自身对齐值为 1，对于 short 型为 2，对于 int,float,double 类型，其自身对齐值为 4，单位字节。
2.结构体或者类的自身对齐值：其成员中自身对齐值最大的那个值。
3.指定对齐值：#pragma pack (value)时的指定对齐值 value。
4.数据成员、结构体和类的有效对齐值：自身对齐值和指定对齐值中小的那个值。

1、 分配内存的顺序是按照声明的顺序。

2、 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，不是整数倍空出内存，直到偏移量是整数倍为止。

3、 最后整个结构体的大小必须是里面变量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：

1、 偏移量要是 n 和当前变量大小中较小值的整数倍

2、 整体大小要是 n 和最大变量大小中较小值的整数倍

3、 n 值必须为 1,2,4,8…，为其他值时就按照默认的分配规则

84、 结构体变量比较是否相等

1. 重载了 “==” 操作符

```
Plaintext
struct foo {  int a;  int b;  bool operator==(const foo& rhs) *//*
*操作运算符重载*  {    return( a == rhs.a) && (b == rhs.b);  }};
```

1. 元素的话，一个个比;
2. 指针直接比较，如果保存的是同一个实例地址，则(p1==p2)为真;

85、 函数调用过程栈的变化，返回值和参数变量哪个先入栈?

1、调用者函数把被调函数所需要的参数按照与被调函数的形参顺序相反的顺序压入栈中,即:从右向左依次把被调

函数所需要的参数压入栈;

2、调用者函数使用 call 指令调用被调函数,并把 call 指令的下一条指令的地址当成返回地址压入栈中(这个压栈操作

隐含在 call 指令中);

3、在被调函数中,被调函数会先保存调用者函数的栈底地址(push ebp),然后再保存调用者函数的栈顶地址,即:当前

被调函数的栈底地址(mov ebp,esp);

4、在被调函数中,从 ebp 的位置处开始存放被调函数中的局部变量和临时变量,并且这些变量的地址按照定义时的

顺序依次减小,即:这些变量的地址是按照栈的延伸方向排列的,先定义的变量先入栈,后定义的变量后入栈;

86、define、const、typedef、inline 的使用方法? 他们之间有什么区别?

一、***const*与#define** 的区别：**

1. const 定义的常量是变量带类型，而#define 定义的只是个常数不带类型;
2. define 只在预处理阶段起作用，简单的文本替换，而 const 在编译、链接过程中起

作用；

3. define 只是简单的字符串替换没有类型检查。而 const 是有数据类型的，是要进行判断的，可以避免一些低级错误；

4. define 预处理后，占用代码段空间，const 占用数据段空间；

5. const 不能重定义，而 define 可以通过#undef 取消某个符号的定义，进行重定义；

6. define 独特功能，比如可以用来防止文件重复引用。

二、 #define 和别名 typedef 的区别

1. 执行时间不同，typedef 在编译阶段有效，typedef 有类型检查的功能；#define 是宏定义，发生在预处理阶段，不进行类型检查；

2. 功能差异，typedef 用来定义类型的别名，定义与平台无关的数据类型，与 struct 的结合使用等。#define 不只是可以为类型取别名，还可以定义常量、变量、编译开关等。

3. 作用域不同，#define 没有作用域的限制，只要是之前预定义过的宏，在以后的程序中都可以使用。而 typedef 有自己的作用域。

三、 define 与 inline 的区别

1. #define 是关键字，inline 是函数；

2. 宏定义在预处理阶段进行文本替换，inline 函数在编译阶段进行替换；

3. inline 函数有类型检查，相比宏定义比较安全；

87、你知道 printf 函数的实现原理是什么吗？

在 C/C++中，对函数参数的扫描是从后向前的。

C/C++的函数参数是通过压入堆栈的方式来给函数传参数的（堆栈是一种先进后出的数据结构），最先压入的参数最后出来，在计算机的内存中，数据有 2 块，一块是堆，一块是栈（函数参数及局部变量在这里），而栈是从内存的高地址向低地址生长的，控制生长的就是堆栈指针了，最先压入的参数是在最上面，就是说在所有参数的最后面，最后压入的参数在最下面，结构上看起来是第一个，所以最后压入的参数总是能够被函数找到，因为它就在堆栈指针的上方。

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.

88、为什么模板类一般都是放在一个 h 文件中

大部分编译器在编译模板时都使用包含模式，也就是一般使用时把模板放到头文件中在包含。当你不使用这个模版函数或模版类,编译器并不实例化它 ,当你使用时，编译器需要实例化它，因为编译器是一次只能处理一个编译单元, 也就是一次处理一个 cpp 文件,所以实例化时需要看到该模板的完整定义，所以都放在头文件中。这不同于普通的函数, 在使用普通的函数时，编译时只需看到该函数的声明即可编译, 而在链接时由链接器来确定该函数的实体。

4.举例说明

```
Plaintext
//test.h
template<class T>
class Test()
{
public:
    mFun();
private:
    T value;
}

_____________________________
//test.cpp
#include"test.h"

void Test<T>::mFun()
{
 ...
}

_____________________________
//main.cpp
#include"test.h"

int main()
{
    Test<int> m_test;
    m_test.mFun();//#1
    return 0;
}
```

编译器在#1 处并不知道 Test::mFun 的定义，因为它不在 test.h 里面，于是编译器只好寄希望于连接器，希望它能够在其他.obj 里面找到 Test::mFun 的实例，在本例中就是 test.obj，然而，后者中真有 Test::mFun 的二进制代码吗？NO!!! 因为 C++标准明确表示，当一个模板不被用到的时侯它就不该被实例化出来，test.cpp 中用到了 Test::mFun 了吗？没有!! 所以实际上 test.cpp 编译出来的 test.obj 文件中关于 Test::mFun 一行二进制代码也没有，于是连接器就傻眼了，只好给出一个连接错误。

但是，如果在 test.cpp 中写一个函数，其中调用 Test::mFun，则编译器会将其实例化出来，因为在这个点上（test.cpp 中），编译器知道模板的定义，所以能够实例化，于是，test.obj 的符号导出表中就有了 Test::mFun 这个符号的地址，于是连接器就能够完成任务。

1. 模板定义很特殊。由 template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。

所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

1. 在分离式编译的环境下，编译器编译某一个.cpp 文件时并不知道另一个.cpp 文件的存在，也不会去查找（当遇到未决符号时它会寄希望于连接器）。这种模式在没有模板的情况下运行良好，但遇到模板时就傻眼了，因为模板仅在需要的时候才会实例化出来。

所以，当编译器只看到模板的声明时，它不能实例化该模板，只能创建一个具有外部连接的符号并期待连接器能够将符号的地址决议出来。

然而当实现该模板的.cpp 文件中没有用到模板的实例时，编译器懒得去实例化，所以，整个工程的.obj 中就找不到一行模板实例的二进制代码，于是连接器也黔驴技穷了。

89、C++中类成员的访问权限和继承权限问题

1. 三种访问权限

2. 一个类的 public 成员变量、成员函数，可以通过类的成员函数、类的实例变量进行访问

3. 一个类的 protected 成员变量、成员函数，无法通过类的实例变量进行访问。但是可以通过类的友元函数、友元类进行访问。

4. 一个类的 private 成员变量、成员函数，无法通过类的实例变量进行访问。但是可以通过类的友元函数、友元类进行访问。

5. 1：public 继承

派生类通过 public 继承，基类的各种权限不变 。

```
Plaintext
 派生类的成员函数，可以访问基类的 public 成员、protected 成员，但是无法访问基类的 private 成员。

 派生类的实例变量，可以访问基类的 public 成员，但是无法访问 protected、private 成员，仿佛基类的成员之间加到了派生类一般。

 可以将 public 继承看成派生类将基类的 public,protected 成员囊括到派生
```

类，但是不包括 private 成员。

1. 2：protected 继承

派生类通过 protected 继承，基类的 public 成员在派生类中的权限变成了 protected 。protected 和 private 不变。
派生类的成员函数，可以访问基类的 public 成员、protected 成员，但是无法访问基类的 private 成员。

```
Plaintext
 派生类的实例变量，无法访问基类的任何成员，因为基类的 public 成员在派生类中变成了 protected。

 可以将 protected 继承看成派生类将基类的 public,protected 成员囊括到派生类，全部作为派生类的 protected 成员，但是不包括 private 成员。

 private 成员是基类内部的隐私，除了友元，所有人员都不得窥探。派生类的友元，都不能访问
```

1. 3：private 继承

```
Plaintext
 派生类通过 private 继承，基类的所有成员在派生类中的权限变成了 private。
 派生类的成员函数，可以访问基类的 public 成员、protected 成员，但是无法访问基类的 private 成员。

 派生类的实例变量，无法访问基类的任何成员，因为基类的所有成员在派生类中变成了 private。

 可以将 private 继承看成派生类将基类的 public,protected 成员囊括到派生类，全部作为派生类的 private 成员，但是不包括 private 成员。

 private 成员是基类内部的隐私，除了友元，所有人员都不得窥探。派生类的友元，都不能访问
```

① public:用该关键字修饰的成员表示公有成员，该成员不仅可以在类内可以被 访问，在类外也是可以被访问的，是类对外提供的可访问接口；

② private:用该关键字修饰的成员表示私有成员，该成员仅在类内可以被访问，在类体外是隐藏状态；

③ protected:用该关键字修饰的成员表示保护成员，保护成员在类体外同样是隐藏状态，但是对于该类的派生类来说，相当于公有成员，在派生类中可以被访问。

1. 三种继承方式

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。

90、cout 和 printf 有什么区别?

cout<<是一个函数，cout<<后可以跟不同的类型是因为 cout<<已存在针对各种类型数据的重载，所以会自动识别数据的类型。输出过程会首先将输出字符放入缓冲区，然后输出到屏幕。

cout 是有缓冲输出:

```
Plaintext
 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 是无缓冲输出。有输出时立即输出

91、你知道重载运算符吗?

赋值运算符= 下表运算符[ ] 函数调用运算符( ) -> 只通过成员函数重载

<< >> 只通过全局函数配合友元函数重载

不要重载 && 和|| 无法实现短路特性

对于内置数据类型，编译器知道如何做运算

前置后置递增运算符

前置效率高点，因为有引用，少一份开销

前置理念：先++，后返回自身

后置理念：先保存原有值，内部++，最后返回临时数据

重载后置++ ，要有占位参数，区分前后置

重载赋值运算符:

系统默认提供赋值运算符只是简单拷贝， 导致类中指向堆区的指针，出现浅拷贝的问题，所以要重载==,

若想链式编程，return *this；返回引用

1、 我们只能重载已有的运算符，而无权发明新的运算符；对于一个重载的运算符，其优先级和结合律与内置类型一致才可以；不能改变运算符操作数个数；

2、 两种重载方式：成员运算符和非成员运算符，成员运算符比非成员运算符少一个参数；下标运算符、箭头运算符必须是成员运算符；

3、 引入运算符重载，是为了实现类的多态性；

4、 当重载的运算符是成员函数时，this 绑定到左侧运算符对象。成员运算符函数的参数数量比运算符对象的数量少一个；至少含有一个类类型的参数；

5、 从参数的个数推断到底定义的是哪种运算符，当运算符既是一元运算符又是二元运算符（+，-，*，&）；

6、 下标运算符必须是成员函数，下标运算符通常以所访问元素的引用作为返回值，同时最好定义下标运算符的常量版本和非常量版本；

7、 箭头运算符必须是类的成员，解引用通常也是类的成员；重载的箭头运算符必须返回类的指针；

92、当程序中有函数重载时，函数的匹配原则和顺序是什么？

1. 名字查找
2. 确定候选函数
3. 寻找最佳匹配

93、定义和声明的区别

如果是指变量的声明和定义： 从编译原理上来说，声明是仅仅告诉编译器，有个某类型的变量会被使用，但是编译器并不会为它分配任何内存。而定义就是分配了内存。

如果是指函数的声明和定义： 声明：一般在头文件里，对编译器说：这里我有一个函数叫 function() 让编译器知道这个函数的存在。 定义：一般在源文件里，具体就是函数的实现过程 写明函数体。

94、全局变量和 static 变量的区别

1、全局变量（外部变量）的说明之前再冠以 static 就构成了静态的全局变量。

全局变量本身就是静态存储方式，静态全局变量当然也是静态存储方式。

这两者在存储方式上并无不同。这两者的区别在于非静态全局变量的作用域是整个源程序，当一个源程序由多个原文件组成时，非静态的全局变量在各个源文件中都是有效的。

而静态全局变量则限制了其作用域，即只在定义该变量的源文件内有效，在同一源程序的其它源文件中不能使用它。由于静态全局变量的作用域限于一个源文件内，只能为该源文件内的函数公用，因此可以避免在其他源文件中引起错误。

static 全局变量与普通的全局变量的区别是 static 全局变量只初始化一次，防止在其他文件单元被引用。

2.static 函数与普通函数有什么区别？ static 函数与普通的函数作用域不同。尽在本文件中。只在当前源文件中使用的函数应该说明为内部函数（static），内部函数应该在当前源文件中说明和定义。

对于可在当前源文件以外使用的函数应该在一个头文件中说明，要使用这些函数的源文件要包含这个头文件。 static 函数与普通函数最主要区别是 static 函数在内存中只有一份，普通静态函数在每个被调用中维持一份拷贝程序的局部变量存在于（堆栈）中，全局变量存在于（静态区）中，动态申请数据存在于（堆）

95、 静态成员与普通成员的区别是什么？

1. 生命周期

静态成员变量从类被加载开始到类被卸载，一直存在；

普通成员变量只有在类创建对象后才开始存在，对象结束，它的生命期结束；

1. 共享方式

静态成员变量是全类共享；普通成员变量是每个对象单独享用的；

1. 定义位置

普通成员变量存储在栈或堆中，而静态成员变量存储在静态全局区；

1. 初始化位置

普通成员变量在类中初始化；静态成员变量在类外初始化；

1. 默认实参

可以使用静态成员变量作为默认实参，

96、说一下你理解的 ifdef endif 代表着什么？

1. 一般情况下，源程序中所有的行都参加编译。但是有时希望对其中一部分内容只在满足一定条件才进行编译，也就是对一部分内容指定编译的条件，这就是“条件编译”。有时，希望当满足某条件时对一组语句进行编译，而当条件不满足时则编译另一组语句。

2. 条件编译命令最常见的形式为：

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

它的作用是：当标识符已经被定义过(一般是用#define 命令定义)，则对程序段 1 进行编译，否则编译程序段 2。 其中#else 部分也可以没有，即：

```
Plaintext
 \#ifdef  程序段 1  \#denif
```

1. 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。

在头文件中使用#define、#ifndef、#ifdef、#endif 能避免头文件重定义。

97、隐式转换，如何消除隐式转换?

1、C++的基本类型中并非完全的对立，部分数据类型之间是可以进行隐式转换的。所谓隐式转换，是指不需要用户干预，编译器私下进行的类型转换行为。很多时候用户可能都不知道进行了哪些转换

2、C++面向对象的多态特性，就是通过父类的类型实现对子类的封装。通过隐式转换，你可以直接将一个子类的对象使用父类的类型进行返回。在比如，数值和布尔类型的转换，整数和浮点数的转换等。某些方面来说，隐式转换给 C++程序开发者带来了不小的便捷。C++是一门强类型语言，类型的检查是非常严格的。

3、 基本数据类型 基本数据类型的转换以取值范围的作为转换基础（保证精度不丢失）。隐式转换发生在从小->大的转换中。比如从 char 转换为 int。从 int->long。自定义对象 子类对象可以隐式的转换为父类对象。

4、 C++中提供了 explicit 关键字，在构造函数声明的时候加上 explicit 关键字，能够禁止隐式转换。

5、如果构造函数只接受一个参数，则它实际上定义了转换为此类类型的隐式转换机制。可以通过将构造函数声明为 explicit 加以制止隐式类型转换，关键字 explicit 只对一个实参的构造函数有效，需要多个实参的构造函数不能用于执行隐式转换，所以无需将这些构造函数指定为 explicit。

98、C++如何处理多个异常的?

1. C++中的异常情况： 语法错误（编译错误）：比如变量未定义、括号不匹配、关键字拼写错误等等编译器在编译时能发现的错误，这类错误可以及时被编译器发现，而且可以及时知道出错的位置及原因，方便改正。 运行时错误：比如数组下标越界、系统内存不足等等。这类错误不易被程序员发现，它能通过编译且能进入运行，但运行时会出错，导致程序崩溃。为了有效处理程序运行时错误，C++中引入异常处理机制

来解决此问题。

2. C++异常处理机制： 异常处理基本思想：执行一个函数的过程中发现异常，可以不用在本函数内立即进行处理， 而是抛出该异常，让函数的调用者直接或间接处理这个问题。 C++异常处理机制由 3 个模块组成：try(检查)、throw(抛出)、catch(捕获) 抛出异常的语句格式为：throw 表达式；如果 try 块中程序段发现了异常则抛出异常。

3. 异常变量的生命周期

```
Plaintext
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 { }
```

99、如何在不使用额外空间的情况下，交换两个数？你有几种方法

```
Plaintext
1) 算术
x = x + y;
y = x - y;
x = x - y;
存在内存溢出状况
2)
异或
x = x^y;// 只能对 int,char..
y = x^y;
x = x^y;
```

100、你知道 strcpy 和 memcpy 的区别是什么吗?

1、复制的内容不同。strcpy 只能复制字符串，而 memcpy 可以复制任意内容，例如字符数组、整型、结构体、类等。 2、复制的方法不同。strcpy 不需要指定长度，它遇到被复制字符的串结束符"\0"才结束，所以容易溢出。memcpy 则是根据其第 3 个参数决定复制的长度。 3、用途不同。通常在复制字符串时用 strcpy，而需要复制其他类型数据时则一般用 memcpy

```
Plaintext
char * strcpy(char * dest, const char * src) // 实现 src 到 dest 的复制
{
    if ((src == NULL) || (dest == NULL)) //判断参数 src 和 dest 的有效性
```

```
    {

           return NULL;
    }
    char *strdest = dest;         //保存目标字符串的首地址
    while ((*strDest++ = *strSrc++)!='\0'); //把 src 字符串的内容复制到 dest 下
    return strdest;
}
```

```
Plaintext
void *memcpy(void *memTo, const void *memFrom, size_t size)
{
    if((memTo == NULL) || (memFrom == NULL)) //memTo 和 memFrom 必须有效
         return NULL;
    char *tempFrom = (char *)memFrom;             //保存 memFrom 首地址
    char *tempTo = (char *)memTo;                  //保存 memTo 首地址
    while(size -- > 0)                //循环 size 次，复制 memFrom 的值到 memTo 中
          *tempTo++ = *tempFrom++ ;
    return memTo;
}
```

101、程序在执行 int main(int argc, char *argv[])时的内存结构，你了解吗？

参数的含义是程序在命令行下运行的时候，需要输入 argc 个参数，每个参数是以 char 类型输入的，依次存在数组里面，数组是 argv[]，所有的参数在指针

char * 指向的内存中，数组的中元素的个数为 argc 个，第一个参数为程序的名称。

```
Plaintext
argc 和 argv 参数在用命令行编译程序时有用。main( int argc, char* argv[], char **env ) 中
第一个参数，int 型的 argc，为整型，用来统计程序运行时发送给 main 函数的命令行参数的个数，在 VS 中默认值为 1。
第二个参数，char*型的 argv[]，为字符串数组，用来存放指向的字符串参数的指针数组，每一个元素指向一个参数。各成员含义如下：
```

```
argv[0]指向程序运行的全路径名
argv[1]指向在 DOS 命令行中执行程序名后的第一个字符串
argv[2]指向执行程序名后的第二个字符串
argv[3]指向执行程序名后的第三个字符串
argv[argc]为 NULL
```

102、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，比如：操作系统、硬件或者其它线程等。遇到这个关键字声明的变量，编译器对访问该变量的代码就不再进行优化，从而可以提供对特殊地址的稳定访问。

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。

volatile 用在如下的几个地方：

1. 中断服务程序中修改的供其它程序检测的变量需要加 volatile;
2. 多任务环境下各任务间共享的标志应该加 volatile;
3. 存储器映射的硬件寄存器通常也要加 volatile 说明，因为每次对它的读写都可能由不同意义;

103、如果有一个空类，它会默认添加哪些函数?

```
Plaintext
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//
```

104、C++中标准库是什么?

1. C++ 标准库可以分为两部分：

标准函数库： 这个库是由通用的、独立的、不属于任何类的函数组成的。函数库继承自 C 语言。

面向对象类库： 这个库是类及其相关函数的集合。

1. 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. 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

105、你知道 const char* 与 string 之间的关系是什么吗?

1. 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

const char* 可隐式转换为 string

string 不可隐式转换为 char* 型

1. 三者的转化关系如下所示:

```
Plaintext
a)  string 转 const
char* string s = “abc”;
const char* c_s = s.c_str();
b)  const char* 转 string，直接赋值即可
const char* c_s = “abc”;
string s(c_s);
c) string 转 char*
string s = “abc”;  char* c;  const int len = s.length();  c = new
char[len+1];  strcpy(c,s.c_str());
d)  char* 转 string  char* c = “abc”;  string s(c); e)  const char*
转 char*  const char* cpc = “abc”;  char* pc = new
char[strlen(cpc)+1];  strcpy(pc,cpc);f)  char* 转 const char*，直接
赋值即可  char* pc = “abc”;  const char* cpc = pc;
```

106、你什么情况用指针当参数，什么时候用引用，为什么?

1. 使用引用参数的主要原因有两个:

程序员能修改调用函数中的数据对象

通过传递引用而不是整个数据–对象，可以提高程序的运行速度

1. 一般的原则: 对于使用引用的值而不做修改的函数:

如果数据对象很小，如内置数据类型或者小型结构，则按照值传递;

如果数据对象是数组，则使用指针（唯一的选择），并且指针声明为指向 const 的指针;

如果数据对象是较大的结构，则使用 const 指针或者引用，已提高程序的效率。这样可以节省结构所需的时间和空间;

如果数据对象是类对象，则使用 const 引用（传递类对象参数的标准方式是按照引用传递）;

1. 对于修改函数中数据的函数：

如果数据是内置数据类型，则使用指针

如果数据对象是数组，则只能使用指针

如果数据对象是结构，则使用引用或者指针

如果数据是类对象，则使用引用

107、你知道静态绑定和动态绑定吗？讲讲？

1. 对象的静态类型：对象在声明时采用的类型。是在编译期确定的。

2. 对象的动态类型：目前所指对象的类型。是在运行期决定的。对象的动态类型可以更改，但是静态类型无法更改。

3. 静态绑定：绑定的是对象的静态类型，某特性（比如函数依赖于对象的静态类型，发生在编译期。)

4. 动态绑定：绑定的是对象的动态类型，某特性（比如函数依赖于对象的动态类型，发生在运行期。)

108、如何设计一个类计算子类的个数?

1、为类设计一个 static 静态变量 count 作为计数器;

2、类定义结束后初始化 count;

3、在构造函数中对 count 进行+1;

4、 设计拷贝构造函数，在进行拷贝构造函数中进行 count +1，操作;

5、设计复制构造函数，在进行复制函数中对 count+1 操作;

6、在析构函数中对 count 进行-1;

109、怎么快速定位错误出现的地方

1、如果是简单的错误，可以直接双击错误列表里的错误项或者生成输出的错误信息中带行号的地方就可以让编辑窗口定位到错误的位置上。

2、对于复杂的模板错误，最好使用生成输出窗口。

多数情况下出发错误的位置是最靠后的引用位置。如果这样确定不了错误，就需要先把自己写的代码里的引用位置找出来，然后逐个分析了。

110、成员初始化列表会在什么时候用到？它的调用过程是什么?

1. 当初始化一个引用成员变量时;

2. 初始化一个 const 成员变量时；

3. 当调用一个基类的构造函数，而构造函数拥有一组参数时；

4. 当调用一个成员类的构造函数，而他拥有一组参数；

5. 编译器会一一操作初始化列表，以适当顺序在构造函数之内安插初始化操作，并且在任何显示用户代码前。list 中的项目顺序是由类中的成员声明顺序决定的，不是初始化列表中的排列顺序决定的。

112、说一说 strcpy、sprintf 与 memcpy 这三个函数的不同之处

1. 操作对象不同

① strcpy 的两个操作对象均为字符串

② sprintf 的操作源对象可以是多种数据类型，目的操作对象是字符串

③ memcpy 的两个对象就是两个任意可操作的内存地址，并不限于何种数据类型。

1. 执行效率不同

memcpy 最高，strcpy 次之，sprintf 的效率最低。

1. 实现功能不同

① strcpy 主要实现字符串变量间的拷贝

② sprintf 主要实现其他数据类型格式到字符串的转化

③ memcpy 主要是内存块间的拷贝。

113、将引用作为函数参数有哪些好处?

1. 传递引用给函数与传递指针的效果是一样的。

这时，被调函数的形参就成为原来主调函数中的实参变量或对象的一个别名来使用，所以在被调函数中对形参变量的操作就是对其相应的目标对象（在主调函数中）的操作。

1. 使用引用传递函数的参数，在内存中并没有产生实参的副本，它是直接对实参操作；

而使用一般变量传递函数的参数，当发生函数调用时，需要给形参分配存储单元，形参变量是实参变量的副本；

如果传递的是对象，还将调用拷贝构造函数。因此，当参数传递的数据较大时，用引用比用一般变量传递参数的效率和所占空间都好。

1. 使用指针作为函数的参数虽然也能达到与使用引用的效果，但是，在被调函数中

同样要给形参分配存储单元，且需要重复使用"*指针变量名"的形式进行运算，这很容易产生错误且程序的阅读性较差；

另一方面，在主调函数的调用点处，必须用变量的地址作为实参。而引用更容易使用，更清晰。

114、你知道数组和指针的区别吗？

1. 数组在内存中是连续存放的，开辟一块连续的内存空间；数组所占存储空间：sizeof（数组名）；数组大小：sizeof(数组名)/sizeof(数组元素数据类型)；

2. 用运算符 sizeof 可以计算出数组的容量（字节数）。sizeof(p),p 为指针得到的是一个指针变量的字节数，而不是 p 所指的内存容量。

3. 编译器为了简化对数组的支持，实际上是利用指针实现了对数组的支持。具体来说，就是将表达式中的数组元素引用转换为指针加偏移量的引用。

4. 在向函数传递参数的时候，如果实参是一个数组，那用于接受的形参为对应的指针。也就是传递过去是数组的首地址而不是整个数组，能够提高效率；

5. 在使用下标的时候，两者的用法相同，都是原地址加上下标值，不过数组的原地址就是数组首元素的地址是固定的，指针的原地址就不是固定的。

115、如何阻止一个类被实例化？有哪些方法？

1. 将类定义为抽象基类或者将构造函数声明为 private；

2. 不允许类外部创建类对象，只能在类内部创建对象

116、 如何禁止程序自动生成拷贝构造函数？

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。

117、你知道 Debug 和 release 的区别是什么吗？

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。

3. 实际上，Debug 和 Release 并没有本质的界限，他们只是一组编译选项的集合，编译器只是按照预定的选项行动。事实上，我们甚至可以修改这些选项，从而得到优化过的调试版本或是带跟踪语句的发布版本。

118、main 函数的返回值有什么值得考究之处吗？

main 函数的返回值用于说明程序的退出状态。如果返回 0，则代表程序正常退出。返回其它数字的含义则由系统决定。通常，返回非零代表程序异常退出。

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的调用者（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。

119、模板会写吗？写一个比较大小的模板函数

```
Plaintext
#include<iostream>
using namespace std;
template<typename type1,typename type2>//函数模板
type1 Max(type1 a,type2 b)
{    return a > b ? a : b; }
void main()
{   cout<<"Max = "<<Max(5.5,'a')<<endl; }
```

120、strcpy 函数和 strncpy 函数的区别？哪个函数更安全？

1. 函数原型

```
Plaintext
char* strcpy(char* strDest, const char* strSrc)
char* strncpy(char* strDest, const char* strSrc, int pos)
```

1. 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。

2. 如果目标长>指定长>源长，则将源长全部拷贝到目标长，自动加上'\0' 如果指定长<源长，则将源长中按指定长度拷贝到目标字符串，不包括'\0' 如果指定长>目标长，运行时错误；

121、static_cast 比 C 语言中的转换强在哪里？

1. 更加安全；

2. 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

122、成员函数里 memset(this,0,sizeof(*this))会发生什么

1. 有时候类里面定义了很多 int,char,struct 等 c 语言里的那些类型的变量，我习惯在构造函数中将它们初始化为 0，但是一句句的写太麻烦，所以直接就 memset(this, 0, sizeof *this);将整个对象的内存全部置为 0。对于这种情形可以很好的工作，但是下面几种情形是不可以这么使用的；

2. 类含有虚函数表：这么做会破坏虚函数表，后续对虚函数的调用都将出现异常；

3. 类中含有 C++类型的对象：例如，类中定义了一个 list 的对象，由于在构造函数体的代码执行之前就对 list 对象完成了初始化，假设 list 在它的构造函数里分配了内存，那么我们这么一做就破坏了 list 对象的内存。

123、你知道回调函数吗？它的作用？

1. 当发生某种事件时，系统或其他函数将会自动调用你定义的一段函数；

2. 回调函数就相当于一个中断处理函数，由系统在符合你设定的条件时自动调用。为此，你需要做三件事：1，声明；2，定义；3，设置触发条件，就是在你的函数中把你的回调函数名称转化为地址作为一个参数，以便于系统调用；

3. 回调函数就是一个通过函数指针调用的函数。如果你把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用为调用它所指向的函数时，我们就说这是回调函数；

4. 因为可以把调用者与被调用者分开。调用者不关心谁是被调用者，所有它需知道的，只是存在一个具有某种特定原型、某些限制条件（如返回值为 int）的被调用函数。

124、什么是一致性哈希？

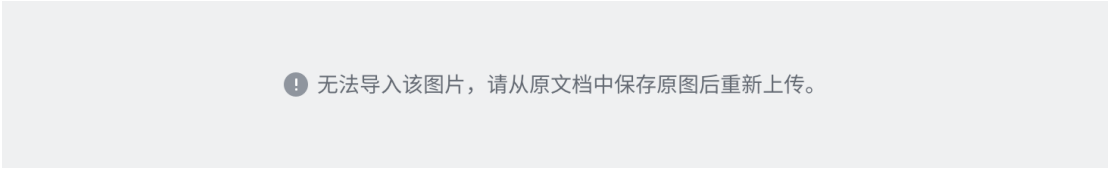
一致性哈希

一致性哈希是一种哈希算法，就是**在移除或者增加一个结点时，能够尽可能小的改变已存在 key 的映射关系**

尽可能少的改变已有的映射关系，一般是沿着顺时针进行操作，回答之前可以先想想，

真实情况如何处理

一致性哈希将整个哈希值空间组**织成一个虚拟的圆环**，假设哈希函数的值空间为0~2^32-1，整个哈希空间环如下左图所示

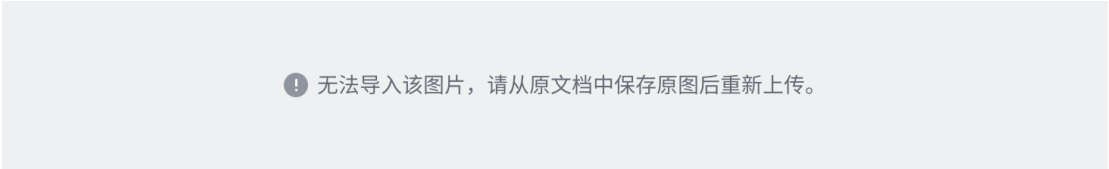


一致性 hash 的基本思想就是使用相同的 hash 算法将数据和结点都映射到图中的环形哈希空间中，上右图显示了 4 个数据 object1-object4 在环上的分布图

结点和数据映射

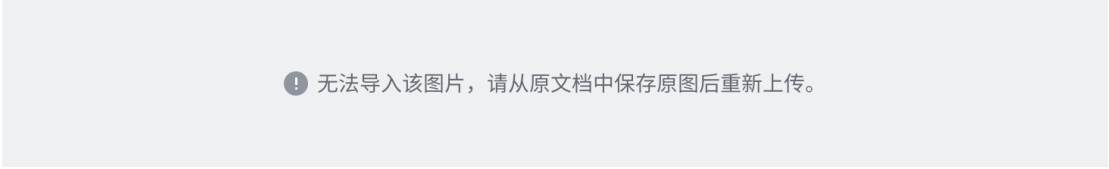
假如有一批服务器，可以根据 IP 或者主机名作为关键字进行哈希，根据结果映射到哈希环中，3 台服务器分别是 nodeA-nodeC

现在有一批的数据 object1-object4 需要存在服务器上，则可以使用相同的哈希算法对数据进行哈希，其结果必然也在环上，可以沿着顺时针方向寻找，找到一个结点（服务器）则将数据存在这个结点上，这样数据和结点就产生了一对一的关联，如下图所示：



移除结点

如果一台服务器出现问题，如上图中的 nodeB，则受影响的是其逆时针方向至下一个结点之间的数据，只需将这些数据映射到它顺时针方向的第一个结点上即可，下左图

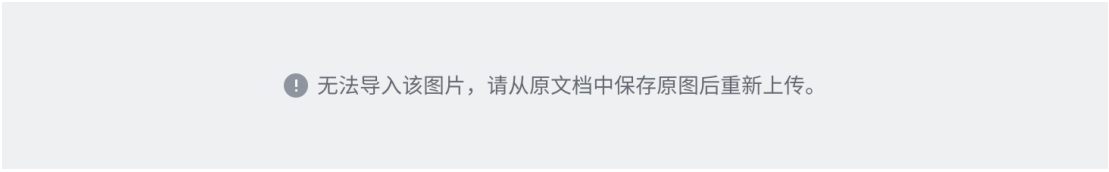


添加结点

如果新增一台服务器 nodeD，受影响的是其逆时针方向至下一个结点之间的数据，将这些数据映射到 nodeD 上即可，见上右图

虚拟结点

假设仅有 2 台服务器：nodeA 和 nodeC，nodeA 映射了 1 条数据，nodeC 映射了 3 条，这样数据分布是不平衡的。引入虚拟结点，假设结点复制个数为 2，则 nodeA 变成：nodeA1 和 nodeA2，nodeC 变成：nodeC1 和 nodeC2，映射情况变成如下：



这样数据分布就均衡多了，平衡性有了很大的提高

125、C++从代码到可执行程序经历了什么？

（1）预编译

主要处理源代码文件中的以“#”开头的预编译指令。处理规则见下：

1. 删除所有的#define，展开所有的宏定义。
2. 处理所有的条件预编译指令，如“#if”、“#endif”、“#ifdef”、“#elif”和“#else”。
3. 处理“#include”预编译指令，将文件内容替换到它的位置，这个过程是递归进行的，文件中包含其他 文件。
4. 删除所有的注释，“//”和“/**/”。
5. 保留所有的#pragma 编译器指令，编译器需要用到他们，如：#pragma once 是为了防止有文件被重 复引用。
6. 添加行号和文件标识，便于编译时编译器产生调试用的行号信息，和编译时产生编译错误或警告是 能够显示行号。

（2）编译

把预编译之后生成的 xxx.i 或 xxx.ii 文件，进行一系列词法分析、语法分析、语义分析及优化后，生成相应 的汇编代码文件。

1. 词法分析：利用类似于“有限状态机”的算法，将源代码程序输入到扫描机中，将其中的字符序列分 割成一系列的记号。
2. 语法分析：语法分析器对由扫描器产生的记号，进行语法分析，产生语法树。由语法分析器输出的 语法树是一种以表达式为节点的树。
3. 语义分析：语法分析器只是完成了对表达式语法层面的分析，语义分析器则对表达式是否有意义进 行判断，其分析的语义是静态语义——在编译期能分期的语义，相对应的动态语义是在运行期才能确定 的语义。
4. 优化：源代码级别的一个优化过程。
5. 目标代码生成：由代码生成器将中间代码转换成目标机器代码，生成一系列的代码序列——汇编语言 表示。
6. 目标代码优化：目标代码优化器对上述的目标机器代码进行优化：寻找合适的寻址方式、使用位移 来替代乘法运算、删除多余的指令等。

（3）汇编

将汇编代码转变成机器可以执行的指令(机器码文件)。 汇编器的汇编过程相对于编译器来说更简单，没 有复杂的语法，也没有语义，更不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译过 来，汇编过程有汇编器 as 完成。经汇编之后，产生目标文件(与可执行文件格式几乎一样)xxx.o(Windows 下)、xxx.obj(Linux 下)。

（4）链接

将不同的源文件产生的目标文件进行链接，从而形成一个可以执行的程序。链接分为静态链接和动态链 接:

静态链接

函数和数据被编译进一个二进制文件。在使用静态库的情况下，在编译链接可执行文件时，链接器从库 中复制这些函数和数据并把它们和应用程序的其它模块组合起来创建最终的可执行文件。

空间浪费：因为每个可执行程序中对所有需要的目标文件都要有一份副本，所以如果多个程序对同一个 目标文件都有依赖，会出现同一个目标文件都在内存存在多个副本;

更新困难：每当库函数的代码修改了，这个时候就需要重新进行编译链接形成可执行程序。

运行速度快：但是静态链接的优点就是，在可执行程序中已经具备了所有执行程序所需要的任何东西， 在执行的时候运行速度快。

动态链接

动态链接的基本思想是把程序按照模块拆分成各个相对独立部分，在程序运行时才将它们链接在一起形 成一个完整的程序，而不是像静态链接一样把所有程序模块都链接成一个单独的可执行文件。

共享库：就是即使需要每个程序都依赖同一个库，但是该库不会像静态链接那样在内存中存在多分，副 本，而是这多个程序在执行时共享同一份副本;

更新方便：更新时只需要替换原来的目标文件，而无需将所有的程序再重新链接一遍。当程序下一次运 行时，新版本的目标文件会被自动加载到内存并且链接起来，程序就完成了升级的目标。

性能损耗：因为把链接推迟到了程序运行时，所以每次执行程序都需要进行链接，所以性能会有一定损 失。

126、为什么友元函数必须在类内部声明?

因为编译器必须能够读取这个结构的声明以理解这个数据类型的大、行为等方面的所有规则。

有一条规则在任何关系中都很重要，那就是谁可以访问我的私有部分。

友元函数和友元类的基本情况

友元提供了不同类的成员函数之间、类的成员函数和一般函数之间进行数据共享的机制。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。友元的正确使用能提高程序的运行效率，但同时也破坏了类的封装性和数据的隐藏性，导致程序可维护性变差。

1）友元函数

有元函数是定义在类外的普通函数，不属于任何类，可以访问其他类的私有成员。但是需要在类的定义中声明所有可以访问它的友元函数。

```
Plaintext
#include <iostream>

using namespace std;

class A
{
public:
    friend void set_show(int x, A &a);      //该函数是友元函数的声明
private:
    int data;
};

void set_show(int x, A &a)  //友元函数定义，为了访问类 A 中的成员
{
    a.data = x;
    cout << a.data << endl;
}
int main(void)
{
    class A a;

    set_show(1, a);

    return 0;
}
```

一个函数可以是多个类的友元函数，但是每个类中都要声明这个函数。

2）友元类

友元类的所有成员函数都是另一个类的友元函数，都可以访问另一个类中的隐藏信息

（包括私有成员和保护成员）。
但是另一个类里面也要相应的进行声明

```
Plaintext
 #include <iostream>

using namespace std;

class A
{
public:
    friend class C;                              //这是友元类的声明
private:
    int data;
};

class C              //友元类定义，为了访问类 A 中的成员
{
public:
    void set_show(int x, A &a) { a.data = x; cout<<a.data<<endl;}
};

int main(void)
{
    class A a;
    class C c;

    c.set_show(1, a);

    return 0;
}
```

使用友元类时注意:

(1) 友元关系不能被继承。

(2) 友元关系是单向的，不具有交换性。若类 B 是类 A 的友元，类 A 不一定是类 B 的友元，要看在类中是否有相应的声明。

(3) 友元关系不具有传递性。若类 B 是类 A 的友元，类 C 是 B 的友元，类 C 不一定是类 A 的友元，同样要看类中是否有相应的申明

127、用 C 语言实现 C++的继承

```
Plaintext
```

```
#include <iostream>

using namespace std;

//C++中的继承与多态
struct A

{

  virtual void fun()  //C++中的多态:通过虚函数实现

  {
    cout<<"A:fun()"<<endl;

  }

  int a;

};

struct B:public A     //C++中的继承:B 类公有继承 A 类

{
  virtual void fun()  //C++中的多态:通过虚函数实现（子类的关键字
virtual 可加可不加）

  {

   cout<<"B:fun()"<<endl;

  }
  int b;

};

//C 语言模拟 C++的继承与多态

typedef void (*FUN)();   //定义一个函数指针来实现对成员函数的继承

struct _A    //父类

{
```

```
  FUN _fun;  //由于 C 语言中结构体不能包含函数，故只能用函数指针在外面实现
  int _a;

};

struct _B     //子类

{

  _A _a_;   //在子类中定义一个基类的对象即可实现对父类的继承

  int _b;

};

void _fA()    //父类的同名函数

{

  printf("_A:_fun()\n");

}

void _fB()    //子类的同名函数

{

  printf("_B:_fun()\n");

}

void Test()

{

  //测试 C++中的继承与多态

  A a;  //定义一个父类对象 a

  B b;  //定义一个子类对象 b
```

```
  A* p1 = &a;  //定义一个父类指针指向父类的对象

  p1->fun();  //调用父类的同名函数

  p1 = &b;   //让父类指针指向子类的对象

  p1->fun();  //调用子类的同名函数


  //C 语言模拟继承与多态的测试

  _A _a;  //定义一个父类对象_a

  _B _b;  //定义一个子类对象_b

  _a._fun = _fA;    //父类的对象调用父类的同名函数

  _b._a_._fun = _fB;  //子类的对象调用子类的同名函数


  _A* p2 = &_a;  //定义一个父类指针指向父类的对象

  p2->_fun();   //调用父类的同名函数

  p2 = (_A*)&_b; //让父类指针指向子类的对象,由于类型不匹配所以要进行强转

  p2->_fun();   //调用子类的同名函数

}
```

128、动态编译与静态编译

1. 静态编译，编译器在编译可执行文件时，把需要用到的对应动态链接库中的部分提取出来，连接到可执行文件中去，使可执行文件在运行时不需要依赖于动态链接库；

2. 动态编译的可执行文件需要附带一个动态链接库，在执行时，需要调用其对应动态链接库的命令。所以其优点一方面是缩小了执行文件本身的体积，另一方面是加快了编译速度，节省了系统资源。缺点是哪怕是很简单的程序，只用到了链接库的一两条命令，也需要附带一个相对庞大的链接库；二是如果其他计算机上没有安装对应的

运行库，则用动态编译的可执行文件就不能运行。

129、hello.c 程序的编译过程

以下是一个 hello.c 程序:

```
Plaintext
#include <stdio.h>int main(){    printf("hello, world\n");
return 0;}
```

在 Unix 系统上，由编译器把源文件转换为目标文件。

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

这个过程大致如下:

- 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。

静态链接

静态链接器以一组可重定位目标文件为输入，生成一个完全链接的可执行目标文件作为输出。链接器主要完成以下两个任务:

- 符号解析：每个符号对应于一个函数、一个全局变量或一个静态变量，符号解析的目的是将每个符号引用与一个符号定义关联起来。
- 重定位：链接器通过把每个符号定义与一个内存位置关联起来，然后修改所有对这些符号的引用，使得它们指向这个内存位置。

目标文件

- 可执行目标文件：可以直接在内存中执行；
- 可重定位目标文件：可与其它可重定位目标文件在链接阶段合并，创建一个可执行目标文件；
- 共享目标文件：这是一种特殊的可重定位目标文件，可以在运行时被动态加载进内存并链接；

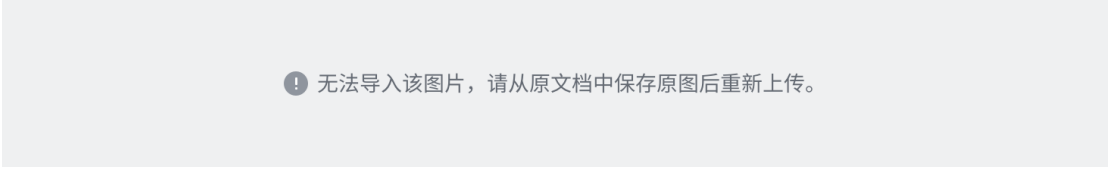
动态链接

静态库有以下两个问题：

- 当静态库更新时那么整个程序都要重新进行链接；
- 对于 printf 这种标准函数库，如果每个程序都要有代码，这会极大浪费资源。

共享库是为了解决静态库的这两个问题而设计的，在 Linux 系统中通常用 .so 后缀来表示，Windows 系统上它们被称为 DLL。它具有以下特点：

- 在给定的文件系统中一个库只有一个文件，所有引用该库的可执行目标文件都共享这个文件，它不会被复制到引用它的可执行文件中；
- 在内存中，一个共享库的 .text 节（已编译程序的机器代码）的一个副本可以被不同的正在运行的进程共享。



源代码－－>预处理－－>编译－－>优化－－>汇编－－>链接-->可执行文件

1. 预处理

读取 c 源程序，对其中的伪指令（以#开头的指令）和特殊符号进行处理。包括宏定义替换、条件编译指令、头文件包含指令、特殊符号。 预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。.i 预处理后的 c 文件，.ii 预处理后的 C++文件。

1. 编译阶段

编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。.s 文件

1. 汇编过程

汇编过程实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个 C 语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。.o 目标文件

1. 链接阶段

链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够诜操作系统装入执行的统一整体。

130、介绍一下几种典型的锁

读写锁

- 多个读者可以同时进行读
- 写者必须互斥（只允许一个写者写，也不能读者写者同时进行）
- 写者优先于读者（一旦有写者，则后续读者必须等待，唤醒时优先考虑写者）

互斥锁

一次只能一个线程拥有互斥锁，其他线程只有等待

互斥锁是在抢锁失败的情况下主动放弃 CPU 进入睡眠状态直到锁的状态改变时再唤醒，而操作系统负责线程调度，为了实现锁的状态发生改变时唤醒阻塞的线程或者进程，需要把锁交给操作系统管理，所以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。而条件变量通过允许线程阻塞和等待另一个线程发送信号的方法弥补了互斥锁的不足，他常和互斥锁一起使用，以免出现竞态条件。当条件不满足时，线程往往解开相应的互斥锁并阻塞线程然后等待条件发生变化。一旦其他的某个线程改变了条件变量，他将通知相应的条件变量唤醒一个或多个正被此条件变量阻塞的线程。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。

131、delete 和 delete[]区别?

- delete 只会调用一次析构函数。
- delete[]会调用数组中每个元素的析构函数。

132、为什么不能把所有的函数写成内联函数?

内联函数以代码复杂为代价，它以省去函数调用的开销来提高执行效率。所以一方面如果内联函数体内代码执行时间相比函数调用开销较大，则没有太大的意义；另一方面每一处内联函数的调用都要复制代码，消耗更多的内存空间，因此以下情况不宜使用内联函数:

- 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

133、为什么 C++没有垃圾回收机制？这点跟 Java 不太一样。

- 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。

134、在进行函数参数以及返回值传递时，可以使用引用或者值传递，其中使用引用的好处有哪些？

对比值传递，引用传参的好处:

1）在函数内部可以对此参数进行修改

2）提高函数调用和运行的效率（因为没有了传值和生成副本的时间和空间消耗）

如果函数的参数实质就是形参，不过这个形参的作用域只是在函数体内部，也就是说实参和形参是两个不同的东西，要想形参代替实参，肯定有一个值的传递。函数调用时，值的传递机制是通过“形参=实参”来对形参赋值达到传值目的，产生了一个实参的副本。即使函数内部有对参数的修改，也只是针对形参，也就是那个副本，实参不会有任何更改。函数一旦结束，形参生命也宣告终结，做出的修改一样没对任何变量产生影响。

用引用作为返回值最大的好处就是在内存中不产生被返回值的副本。

但是有以下的限制:

1）不能返回局部变量的引用。因为函数返回以后局部变量就会被销毁

2）不能返回函数内部 new 分配的内存的引用。虽然不存在局部变量的被动销毁问题，可对于这种情况（返回函数内部 new 分配内存的引用），又面临其它尴尬局面。例如，被函数返回的引用只是作为一 个临时变量出现，而没有被赋予一个实际的变量，那么这个引用所指向的空间（由 new 分配）就无法释放，造成 memory leak

3）可以返回类成员的引用，但是最好是 const。因为如果其他对象可以获得该属性的非常量的引用，那么对该属性的单纯赋值就会破坏业务规则的完整性。

2.1.2、内存管理

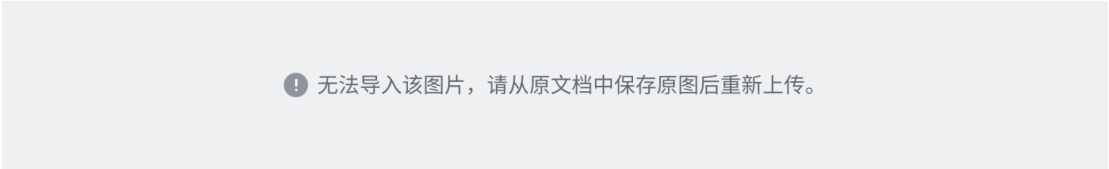
1、类的对象存储空间?

- 非静态成员的数据类型大小之和。
- 编译器加入的额外成员变量（如指向虚函数表的指针）。
- 为了边缘对齐优化加入的 padding。

空类(无非静态数据成员)的对象的 size 为 1, 当作为基类时, size 为 0.

2、简要说明 C++的内存分区

C++中的内存分区，分别是堆、栈、自由存储区、全局/静态存储区、常量存储区和代码区。如下图所示



栈：在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限

堆：就是那些由 `new` 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 `new` 就要对应一个 `delete`。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收

自由存储区：就是那些由 `malloc` 等分配的内存块，它和堆是十分相似的，不过它是用 `free` 来结束自己的生命的

全局/静态存储区：全局变量和静态变量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：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

代码区：存放函数体的二进制代码

3、什么是内存池，如何实现

内存池（Memory Pool） 是一种**内存分配**方式。通常我们习惯直接使用 new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。内存池则是在真正使用内存之前，先申请分配一

定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块， 若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

这里简单描述一下《STL 源码剖析》中的内存池实现机制：

allocate 包装 malloc,deallocate 包装 free

一般是一次 20*2 个的申请，先用一半，留着一半，为什么也没个说法，侯捷在 STL 那边书里说好像是 C++委员会成员认为 20 是个比较好的数字，既不大也不小

1. 首先客户端会调用 malloc()配置一定数量的区块（固定大小的内存块，通常为 8 的倍数），假设 40 个 32bytes 的区块，其中 20 个区块（一半）给程序实际使用，1 个区块交出，另外 19 个处于维护状态。剩余 20 个（一半）留给内存池，此时一共有（20*32byte）

2. 客户端之后有有内存需求，想申请（20*64bytes）的空间，这时内存池只有（20*32bytes），就先将（10*64bytes)个区块返回，1 个区块交出，另外 9 个处于维护状态，此时内存池空空如也

3. 接下来如果客户端还有内存需求，就必须再调用 malloc()配置空间，此时新申请的区块数量会增加一个随着配置次数越来越大的附加量，同样一半提供程序使用，另一半留给内存池。申请内存的时候用永远是先看内存池有无剩余，有的话就用上，然后挂在 0-15 号某一条链表上，要不然就重新申请。

4. 如果整个堆的空间都不够了，就会在原先已经分配区块中寻找能满足当前需求的区块数量，能满足就返回，不能满足就向客户端报 bad_alloc 异常

allocator 就是用来分配内存的，最重要的两个函数是 allocate 和 deallocate，就是用来申请内存和回收内存的，外部（一般指容器）调用的时候只需要知道这些就够了。内部实现，目前的所有编译器都是直接调用的::operator new()和::operator delete()，说白了就是和直接使用 new 运算符的效果是一样的，所以老师说它们都没做任何特殊处理。

最开始 GC2.9 之前

new 和 operator new 的区别：new 是个运算符，编辑器会调用 operator new(0)

operator new()里面有调用 malloc 的操作，那同样的 operator delete()里面有调用的 free 的操作

GC2.9 的 alloc 的一个比较好的分配器的实现规则

维护一条 0-15 号的一共 16 条链表，其中 0 表示 8 bytes ， 1 表示 16 bytes,2 表示 24bytes。。。。而 15 表示 16* 8 = 128bytes，如果在申请时并不是 8 的倍数，那就找刚好能满足内存大小的那个位置。比如想申请 12，那就是找 16 了，想申请 20 ，那就找 24 了

但是现在 GC4.9 及其之后 也还有，只不过已经变成_pool_alloc 这个名字了，不再是默认的了，你需要自己去指定它可以自己指定，比如说 vector<string,__gnu_cxx::pool_alloc> vec;这样来使用它，现在用的又回到以前那种对 malloc 和 free 的包装形式了

4、可以说一下你了解的 C++得内存管理吗?

在 C++中，内存分成 5 个区，他们分别是堆、栈、全局/静态存储区和常量存储区和代码区。

- 栈，在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- 堆，就是那些由 new 分配的内存块，他们的释放编译器不去管，由我们的应用程序去控制，一般一个 new 就要对应一个 delete。如果程序员没有释放掉，那么在程序结束后，操作系统会自动回收。
- 全局/静态存储区，内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。它主要存放静态数据（局部 static 变量，全局 static 变量）、全局变量和常量。
- 常量存储区，这是一块比较特殊的存储区，他们里面存放的是常量字符串，不允许修改。
- 代码区，存放程序的二进制代码

5、C++中类的数据成员和成员函数内存分布情况

C++类是由结构体发展得来的，所以他们的成员变量（C 语言的结构体只有成员变量）的内存分配机制是一样的。下面我们以类来说明问题，如果类的问题通了，结构体也也就没问题啦。 类分为成员变量和成员函数，我们先来讨论成员变量。

一个类对象的地址就是类所包含的这一片内存空间的首地址，这个首地址也就对应具体某一个成员变量的地址。(在定义类对象的同时这些成员变量也就被定义了)，举个例子:

```
Plaintext
#include <iostream>using namespace std;class Person{public:
Person()    {        this->age = 23;    }    void printAge()
{        cout << this->age <<endl;    }    ~Person(){}public:
int age;};int main(){    Person p;    cout << "对象地址："<< &p
<<endl;    cout << "age 地址："<< &(p.age) <<endl;    cout << "对象
大小："<< sizeof(p) <<endl;    cout << "age 大小："<< sizeof(p.age)
<<endl;    return 0;}//输出结果//对象地址：0x7fffec0f15a8//age 地址：
```

```
0x7fffec0f15a8//对象大小：4//age 大小：4
```

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象的内存。这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。要是成员函数占用类的对象空间，那么将是多么可怕的事情：定义一次类对象就有成员函数占用一段空间。 我们再来补充一下静态成员函数的存放问题：**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，就像我前面提到的，**所有函数都存放在代码区，静态函数也不例外。所有有人一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。**

6、关于 this 指针你知道什么？全说出来

- this 指针是类的指针，指向对象的首地址。
- this 指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用 this。
- this 指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

this 指针的用处

一个对象的 this 指针并不是对象本身的一部分，不会影响 sizeof(对象)的结果。this 作用域是在类内部，当在类的**非静态成员函数**中访问类的**非静态成员**的时候（全局函数，静态函数中不能使用 this 指针），编译器会自动将对象本身的地址作为一个隐含参数传递给函数。也就是说，即使你没有写上 this 指针，编译器在编译的时候也是加上 this 的，它作为非静态成员函数的隐含形参，对各成员的访问均通过 this 进行

this 指针的使用

一种情况就是，在类的非静态成员函数中返回类对象本身的时候，直接使用 return *this;

另外一种情况是当形参数与成员变量名相同时用于区分，如 this->n = n （不能写成 n = n）

类的 this 指针有以下特点

(1）**this** 只能在成员函数中使用，全局函数、静态函数都不能使用 this。实际上，**成员函数默认第一个参数**为 **T * const this**

如：

```
Plaintext
class A{public:        int func(int p){}};
```

其中，**func** 的原型在编译器看来应该是：

int func(A * const this,int p);

（2）由此可见，**this** 在成员函数的开始前构造，在成员函数的结束后清除。这个生命周期同任何一个函数的参数是一样的，没有任何区别。当调用一个类的成员函数时，编译器将类的指针作为函数的 this 参数传递进去。如：

```
Plaintext
A a;a.func(10);//此处，编译器将会编译成：A::func(&a,10);
```

看起来和静态函数没差别，对吗？不过，区别还是有的。编译器通常会对 this 指针做一些优化，因此，this 指针的传递效率比较高，例如 VC 通常是通过 ecx（计数寄存器）传递 this 参数的。

7、几个 this 指针的易混问题

A. this 指针是什么时候创建的？

this 在成员函数的开始执行前构造，在成员的执行结束后清除。

但是如果 class 或者 struct 里面没有方法的话，它们是没有构造函数的，只能当做 C 的 struct 使用。采用 TYPE xx 的方式定义的话，在栈里分配内存，这时候 this 指针的值就是这块内存的地址。采用 new 的方式创建对象的话，在堆里分配内存，new 操作符通过 eax（累加寄存器）返回分配的地址，然后设置给指针变量。之后去调用构造函数（如果有构造函数的话），这时将这个内存块的地址传给 ecx，之后构造函数里面怎么处理请看上面的回答

B. this 指针存放在何处？堆、栈、全局变量，还是其他？

this 指针会因编译器不同而有不同的放置位置。可能是栈，也可能是寄存器，甚至全局变量。在汇编级别里面，一个值只会以 3 种形式出现：立即数、寄存器值和内存变量值。不是存放在寄存器就是存放在内存中，它们并不是和高级语言变量对应的。

C. this 指针是如何传递类中的函数的？绑定？还是在函数参数的首参数就是 this 指针？那么，this 指针又是如何找到“类实例后函数的”？

大多数编译器通过 ecx（寄数寄存器）寄存器传递 this 指针。事实上，这也是一个潜规则。一般来说，不同编译器都会遵从一致的传参规则，否则不同编译器产生的 obj 就无法匹配了。

在 call 之前，编译器会把对应的对象地址放到 eax 中。this 是通过函数参数的首参来传递的。this 指针在调用之前生成，至于“类实例后函数”，没有这个说法。类在实例化时，只分配类中的变量空间，并没有为函数分配空间。自从类的函数定义完成后，它就在那儿，不会跑的

D. this 指针是如何访问类中的变量的？

如果不是类，而是结构体的话，那么，如何通过结构指针来访问结构中的变量呢？如果你明白这一点的话，就很容易理解这个问题了。

在 C++中，类和结构是只有一个区别的：类的成员默认是 private，而结构是 public。

this 是类的指针，如果换成结构体，那 this 就是结构的指针了。

E.我们只有获得一个对象后，才能通过对象使用 this 指针。如果我们知道一个对象 this 指针的位置，可以直接使用吗?

this 指针只有在成员函数中才有定义。因此，你获得一个对象后，也不能通过对象使用 this 指针。所以，我们无法知道一个对象的 this 指针的位置（只有在成员函数里才有 this 指针的位置）。当然，在成员函数里，你是可以知道 this 指针的位置的（可以通过&this 获得），也可以直接使用它。

F.每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数?

普通的类函数（不论是成员函数，还是静态函数）都不会创建一个函数表来保存函数指针。只有虚函数才会被放到函数表中。但是，即使是虚函数，如果编译期就能明确知道调用的是哪个函数，编译器就不会通过函数表中的指针来间接调用，而是会直接调用该函数。正是由于 this 指针的存在，用来指向不同的对象，从而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？如何监测？解决方法？

1) 内存泄漏

内存泄漏是指由于疏忽或错误造成了程序未能释放掉不再使用的内存的情况。内存泄漏并非指内存在物理上消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制；

2) 后果

只发生一次小的内存泄漏可能不被注意，但泄漏大量内存的程序将会出现各种证照：性能下降到内存逐渐用完，导致另一个程序失败；

3) 如何排除

使用工具软件 BoundsChecker，BoundsChecker 是一个运行时错误检测工具，它主要定位程序运行时期发生的各种错误；

调试运行 DEBUG 版程序，运用以下技术：CRT(C run-time libraries)、运行时函数调用堆栈、内存泄漏时提示的内存分配序号(集成开发环境 OUTPUT 窗口)，综合分析内存泄漏的原因，排除内存泄漏。

4) 解决方法

智能指针。

1. 检查、定位内存泄漏

检查方法：在 main 函数最后面一行，加上一句_CrtDumpMemoryLeaks()。调试程序，自然关闭程序让其退出，查看输出：

输出这样的格式{453}normal block at 0x02432CA8,868 bytes long

被{}包围的 453 就是我们需要的内存泄漏定位值，868 bytes long 就是说这个地方有 868 比特内存没有释放。

定位代码位置

在 main 函数第一行加上_CrtSetBreakAlloc(453);意思就是在申请 453 这块内存的位置中断。然后调试程序，程序中断了，查看调用堆栈。加上头文件#include <crtdbg.h>

9、在成员函数中调用 delete this 会出现什么问题？对象还可以使用吗？

1、在类对象的内存空间中，只有数据成员和虚函数表指针，并不包含代码内容，类的成员函数单独放在代码段中。在调用成员函数时，隐含传递一个 this 指针，让成员函数知道当前是哪个对象在调用它。当调用 delete this 时，类对象的内存空间被释放。在 delete this 之后进行的其他任何函数调用，只要不涉及到 this 指针的内容，都能够正常运行。一旦涉及到 this 指针，如操作数据成员，调用虚函数等，就会出现不可预期的问题。

10、为什么是不可预期的问题？

delete this 之后不是释放了类对象的内存空间了么，那么这段内存应该已经还给系统，不再属于这个进程。照这个逻辑来看，应该发生指针错误，无访问权限之类的令系统崩溃的问题才对啊？这个问题牵涉到操作系统的内存管理策略。delete this 释放了类对象的内存空间，但是内存空间却并不是马上被回收到系统中，可能是缓冲或者其他什么原因，导致这段内存空间暂时并没有被系统收回。此时这段内存是可以访问的，你可以加上 100，加上 200，但是其中的值却是不确定的。当你获取数据成员，可能得到的是一串很长的未初始化的随机数；访问虚函数表，指针无效的可能性非常高，造成系统崩溃。

11、 如果在类的析构函数中调用 delete this，会发生什么？

会导致堆栈溢出。原因很简单，delete 的本质是“为将被释放的内存调用一个或多个析构函数，然后，释放内存”。显然，delete this 会去调用本对象的析构函数，而析构函数中又调用 delete this，形成无限递归，造成堆栈溢出，系统崩溃。

12、你知道空类的大小是多少吗？

首先，我们要知道，一个类是没有大小的，应该说成是类的实例的大小。

1. C++空类的大小不为 0，不同编译器设置不一样，vs 设置为 1；

2. C++标准指出，不允许一个对象（当然包括类对象）的大小为 0，不同的对象不能具有相同的地址；

3. 带有虚函数的 C++类大小不为 1，因为每一个对象会有一个 vptr 指向虚函数表，具体大小根据指针大小确定；

4. C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小，这样便保证了每个实例均有独一无二的内存地址。

13、请说一下以下几种情况下，下面几个类的大小各是多少？

```
Plaintext
class A {};int main(){  cout<<sizeof(A)<<endl;// 输出 1;  A a;
cout<<sizeof(a)<<endl;// 输出 1;  return 0;}
```

空类的大小是 1， 在 C++中空类会占一个字节，这是为了让对象的实例能够相互区别。具体来说，空类同样可以被实例化，并且每个实例在内存中都有独一无二的地址，因此，编译器会给空类隐含加上一个字节，这样空类实例化之后就会拥有独一无二的内存地址。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。

空类的实例大小就是类的大小，所以 sizeof(a)=1 字节,如果 a 是指针，则 sizeof(a)就是指针的大小，即 4 字节。

```
Plaintext
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;}
```

因为有虚函数的类对象中都有一个虚函数表指针 __vptr，其大小是 4 字节

```
Plaintext
class A { static int a; };int main(){  cout<<sizeof(A)<<endl;// 输
出 1;  A a;  cout<<sizeof(a)<<endl;// 输出 1;  return 0;}
```

静态成员存放在静态存储区，不占用类的大小, 普通函数也不占用类大小

```
Plaintext
class A { int a; };int main(){  cout<<sizeof(A)<<endl;// 输出 4;
A a;  cout<<sizeof(a)<<endl;// 输出 4;  return 0;}
```

```
class A { static int a; int b; };;int
main(){  cout<<sizeof(A)<<endl;// 输出 4;  A a;
cout<<sizeof(a)<<endl;// 输出 4;  return 0;}
```

静态成员 a 不占用类的大小，所以类的大小就是 b 变量的大小 即 4 个字节

14、this 指针调用成员变量时，堆栈会发生什么变化?

当在类的非静态成员函数访问类的非静态成员时，编译器会自动将对象的地址传给作为隐含参数传递给函数，这个隐含参数就是 this 指针。

即使你并没有写 this 指针，编译器在链接时也会加上 this 的，对各成员的访问都是通过 this 的。

例如你建立了类的多个对象时，在调用类的成员函数时，你并不知道具体是哪个对象在调用，此时你可以通过查看 this 指针来查看具体是哪个对象在调用。This 指针首先入栈，然后成员函数的参数从右向左进行入栈，最后函数返回地址入栈。

15、类对象的大小受哪些因素影响?

1. 类的非静态成员变量大小，静态成员不占据类的空间，成员函数也不占据类的空间大小;
2. 内存对齐另外分配的空间大小，类内的数据也是需要进行内存对齐操作的;
3. 虚函数的话，会在类对象插入 vptr 指针，加上指针大小;
4. 当该该类是某类的派生类，那么派生类继承的基类部分的数据成员也会存在在派生类中的空间中，也会对派生类进行扩展。

2.1.3、C++11 新标准

1、C++ 11 有哪些新特性?

- nullptr 替代 NULL
- 引入了 auto 和 decltype 这两个关键字实现了类型推导
- 基于范围的 for 循环 for(auto& i : res){}
- 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
- ...

2、auto、decltype 和 decltype(auto)的用法

（1）auto

C++11 新标准引入了 auto 类型说明符，用它就能让编译器替我们去分析表达式所属的类型。和原来那些只对应某种特定的类型说明符(例如 int)不同，

auto 让编译器通过初始值来进行类型推演。从而获得定义变量的类型，所以说 auto 定义的变量必须有初始值。举个例子：

```
Plaintext
//普通；类型
int a = 1, b = 3;
auto c = a + b;// c 为 int 型

//const 类型
const int i = 5;
auto j = i; // 变量 i 是顶层 const，会被忽略，所以 j 的类型是 int
auto k = &i; // 变量 i 是一个常量，对常量取地址是一种底层 const，所以 b 的类型是 const int*
const auto l = i; //如果希望推断出的类型是顶层 const 的，那么就需要在 auto 前面加上 cosnt

//引用和指针类型
int x = 2;
int& y = x;
auto z = y; //z 是 int 型不是 int& 型
auto& p1 = y; //p1 是 int&型
auto p2 = &x; //p2 是指针类型 int*
```

（2）decltype

有的时候我们还会遇到这种情况，**我们希望从表达式中推断出要定义变量的类型，但却不想用表达式的值去初始化变量。**还有可能是函数的返回类型为某表达式的值类型。在这些时候 auto 显得就无力了，所以 C++11 又引入了第二种类型说明符 decltype，**它的作用是选择并返回操作数的数据类型。在此过程中，编译器只是分析表达式并得到它的类型，却不进行实际的计算表达式的值。**

```
Plaintext
int func() {return 0};//普通类型 decltype(func()) sum = 5; // sum 的类型是函数 func()的返回值的类型 int，但是这时不会实际调用函数 func()int a = 0;decltype(a) b = 4; // a 的类型是 int，所以 b 的类型也是 int//不论是顶层 const 还是底层 const，decltype 都会保留    const int c =
```

```
3;decltype(c) d = c; // d 的类型和 c 是一样的，都是顶层 constint e = 4;const int* f = &e; // f 是底层 constdecltype(f) g = f; // g 也是底层 const//引用与指针类型//1. 如果表达式是引用类型，那么 decltype 的类型也是引用 const int i = 3, &j = i;decltype(j) k = 5; // k 的类型是 const int&//2. 如果表达式是引用类型，但是想要得到这个引用所指向的类型，需要修改表达式:int i = 3, &r = i;decltype(r + 0) t = 5; // 此时是 int 类型//3. 对指针的解引用操作返回的是引用类型 int i = 3, j = 6, *p = &i;decltype(*p) c = j; // c 是 int&类型，c 和 j 绑定在一起//4. 如果一个表达式的类型不是引用，但是我们需要推断出引用，那么可以加上一对括号，就变成了引用类型了 int i = 3;decltype((i)) j = i; // 此时 j 的类型是 int&类型，j 和 i 绑定在了一起
```

（3）decltype(auto)

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“=”号左边的表达式替换掉 auto，再根据 decltype 的语法规则来确定类型。举个例子:

```
Plaintext
int e = 4;const int* f = &e; // f 是底层 constdecltype(auto) j = f;//j 的类型是 const int* 并且指向的是 e
```

3、C++中 NULL 和 nullptr 区别

算是为了与 C 语言进行兼容而定义的一个问题吧

NULL 来自 C 语言，一般由宏定义实现，而 nullptr 则是 C++11 的新增关键字。**在 C 语言中，NULL 被定义为(void*)0,而在 C++语言中，NULL 则被定义为整数 0**。编译器一般对其实际定义如下:

```
Plaintext
#ifdef __cplusplus#define NULL 0#else#define NULL ((void *)0)#endif
```

在 C++中指针必须有明确的类型定义。但是将 NULL 定义为 0 带来的另一个问题是无法与整数的 0 区分。因为 C++中允许有函数重载，所以可以试想如下函数定义情况:

```
Plaintext
#include <iostream>using namespace std;void fun(char* p)
{        cout << "char*" << endl;}void fun(int p) {        cout << "int" << endl;}int main(){        fun(NULL);        return 0;}//输出结果：int
```

那么在传入 NULL 参数时，会把 NULL 当做整数 0 来看，如果我们想调用参数是指针的函数，该怎么办呢?。nullptr 在 C++11 被引入用于解决这一问题，nullptr 可以明确区分整型和指针类型，能够根据环境自动转换成相应的指针类型，但不会被转换为任何整型，所以不会造成参数传递错误。

nullptr 的一种实现方式如下：

```
Plaintext
const class nullptr_t{public:    template<class T>  inline
operator T*() const{ return 0; }    template<class C, class T>
inline operator T C::*() const { return 0; }private:    void
operator&() const;} nullptr = {};
```

以上通过模板类和运算符重载的方式来对不同类型的指针进行实例化从而解决了(void*)指针带来参数类型不明的问题，**另外由于 nullptr 是明确的指针类型，所以不会与整形变量相混淆。**但 nullptr 仍然存在一定问题，例如：

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

在这种情况下存在对不同指针类型的函数重载，此时如果传入 nullptr 指针则仍然存在无法区分应实际调用哪个函数，这种情况下必须显示的指明参数类型。

4、智能指针的原理、常用的智能指针及实现

原理

智能指针是一个类，用来存储指向动态分配对象的指针，负责自动释放动态分配的对象，防止堆内存泄漏。动态分配的资源，交给一个类对象去管理，当类对象声明周期结束时，自动调用析构函数释放资源

常用的智能指针

(1) shared_ptr

实现原理：采用引用计数器的方法，允许多个智能指针指向同一个对象，每当多一个指针指向该对象时，指向该对象的所有智能指针内部的引用计数加 1，每当减少一个智能指针指向对象时，引用计数会减 1，当计数为 0 的时候会自动的释放动态分配的资源。

- 智能指针将一个计数器与类指向的对象相关联，引用计数器跟踪共有多少个类对

象共享同一指针

- 每次创建类的新对象时，初始化指针并将引用计数置为 1
- 当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数
- 对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数
- 调用析构函数时，构造函数减少引用计数（如果引用计数减至 0，则删除基础对象）

(2) unique_ptr

unique_ptr 采用的是独享所有权语义，一个非空的 unique_ptr 总是拥有它所指向的资源。转移一个 unique_ptr 将会把所有权全部从源指针转移给目标指针，源指针被置空；所以 unique_ptr 不支持普通的拷贝和赋值操作，不能用在 STL 标准容器中；局部变量的返回值除外（因为编译器知道要返回的对象将要被销毁）；如果你拷贝一个 unique_ptr，那么拷贝结束后，这两个 unique_ptr 都会指向相同的资源，造成在结束时对同一内存指针多次释放而导致程序崩溃。

(3) weak_ptr

weak_ptr：弱引用。 引用计数有一个问题就是互相引用形成环（环形引用），这样两个指针指向的内存都无法释放。需要使用 weak_ptr 打破环形引用。weak_ptr 是一个弱引用，它是为了配合 shared_ptr 而引入的一种智能指针，它指向一个由 shared_ptr 管理的对象而不影响所指对象的生命周期，也就是说，它只引用，不计数。如果一块内存被 shared_ptr 和 weak_ptr 同时引用，当所有 shared_ptr 析构了之后，不管还有没有 weak_ptr 引用该内存，内存也会被释放。所以 weak_ptr 不保证它指向的内存一定是有效的，在使用之前使用函数 lock()检查 weak_ptr 是否为空指针。

(4) auto_ptr

主要是为了解决“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。

auto_ptr 有拷贝语义，拷贝后源对象变得无效，这可能引发很严重的问题；而 unique_ptr 则无拷贝语义，但提供了移动语义，这样的错误不再可能发生，因为很明显必须使用 std::move()进行转移。

auto_ptr 不支持拷贝和赋值操作，不能用在 STL 标准容器中。STL 容器中的元素经常要支持拷贝、赋值操作，在这过程中 auto_ptr 会传递所有权，所以不能在 STL 中使用。

智能指针 shared_ptr 代码实现：

```
Plaintext
template<typename T>
class SharedPtr{
```

```
public:
    SharedPtr(T* ptr = NULL):_ptr(ptr), _pcount(new int(1))
{}                          SharedPtr(const SharedPtr&
s):_ptr(s._ptr), _pcount(s._pcount)
{                                  (*_pcount)++;
    }
    SharedPtr<T>& operator=(const SharedPtr& s){
        if (this != &s)
        {
            if (--(*(this->_pcount)) == 0)
            {
                delete this->_ptr;
                delete this->_pcount;
            }
            _ptr = s._ptr;
            _pcount = s._pcount;
            *(_pcount)++;
        }
        return *this;
   }
   T& operator*() { return *(this->_ptr); }
   T* operator->() { return this->_ptr; }
   ~SharedPtr() {
       --(*(this->_pcount));
       if (*(this->_pcount) == 0)
       {
           delete _ptr;
           _ptr = NULL;
           delete _pcount;
           _pcount = NULL;
       }
    }
private:
    T* _ptr;
    int* _pcount;//指向引用计数的指针
};
```

update1:微信好友“健康哥”指出代码实现部分笔误，感谢。

5、说一说你了解的关于 lambda 函数的全部知识

1. 利用 lambda 表达式可以编写内嵌的匿名函数，用以替换独立函数或者函数对象；

2. 每当你定义一个 lambda 表达式后，编译器会自动生成一个匿名类（这个类当然重载了()运算符），我们称为闭包类型（closure type）。那么在运行时，这个 lambda 表

达式就会返回一个匿名的闭包实例，其实一个右值。所以，我们上面的 lambda 表达式的结果就是一个个闭包。闭包的一个强大之处是其可以通过传值或者引用的方式捕捉其封装作用域内的变量，前面的方括号就是用来定义捕捉模式以及变量，我们又将其称为 lambda 捕捉块。

3. lambda 表达式的语法定义如下：

[capture] （parameters） mutable ->return-type {statement};

1. lambda 必须使用尾置返回来指定返回类型，可以忽略参数列表和返回值，但必须永远包含捕获列表和函数体；

6、智能指针的作用；

1. C++11 中引入了智能指针的概念，方便管理堆内存。使用普通指针，容易造成堆内存泄露（忘记释放），二次释放，程序发生异常时内存泄露等问题等，使用智能指针能更好的管理堆内存。

2. 智能指针在 C++11 版本之后提供，包含在头文件中，shared_ptr、unique_ptr、weak_ptr。shared_ptr 多个指针指向相同的对象。shared_ptr 使用引用计数，每一个 shared_ptr 的拷贝都指向相同的内存。每使用他一次，内部的引用计数加 1，每析构一次，内部的引用计数减 1，减为 0 时，自动删除所指向的堆内存。shared_ptr 内部的引用计数是线程安全的，但是对象的读取需要加锁。

3. 初始化。智能指针是个模板类，可以指定类型，传入指针通过构造函数初始化。也可以使用 make_shared 函数初始化。不能将指针直接赋值给一个智能指针，一个是类，一个是指针。例如 std::shared_ptr p4 = new int(1);的写法是错误的

拷贝和赋值。拷贝使得对象的引用计数增加 1，赋值使得原对象引用计数减 1，当计数为 0 时，自动释放内存。后来指向的对象引用计数加 1，指向后来的对象

1. unique_ptr“唯一”拥有其所指对象，同一时刻只能有一个 unique_ptr 指向给定对象（通过禁止拷贝语义、只有移动语义来实现）。相比与原始指针 unique_ptr 用于其 RAII 的特性，使得在出现异常的情况下，动态资源能得到释放。unique_ptr 指针本身的生命周期：从 unique_ptr 指针创建时开始，直到离开作用域。离开作用域时，若其指向对象，则将其所指对象销毁(默认使用 delete 操作符，用户可指定其他操作)。unique_ptr 指针与其所指对象的关系：在智能指针生命周期内，可以改变智能指针所指对象，如创建智能指针时通过构造函数指定、通过 reset 方法重新指定、通过 release 方法释放所有权、通过移动语义转移所有权。

2. 智能指针类将一个计数器与类指向的对象相关联，引用计数跟踪该类有多少个对象共享同一指针。每次创建类的新对象时，初始化指针并将引用计数置为 1；当对象作为另一对象的副本而创建时，拷贝构造函数拷贝指针并增加与之相应的引用计数；对一个对象进行赋值时，赋值操作符减少左操作数所指对象的引用计数（如果引用计数为减至 0，则删除对象），并增加右操作数所指对象的引用计数；调用析构函数时，

构造函数减少引用计数（如果引用计数减至 0，则删除基础对象）。

3. weak_ptr 是一种不控制对象生命周期的智能指针，它指向一个 shared_ptr 管理的对象. 进行该对象的内存管理的是那个强引用的 shared_ptr. weak_ptr 只是提供了对管理对象的一个访问手段。weak_ptr 设计的目的是为配合 shared_ptr 而引入的一种智能指针来协助 shared_ptr 工作，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.

7、说说你了解的 auto_ptr 作用

1. auto_ptr 的出现，主要是为了解决“有异常抛出时发生内存泄漏”的问题；抛出异常，将导致指针 p 所指向的空间得不到释放而导致内存泄漏；

2. auto_ptr 构造时取得某个对象的控制权，在析构时释放该对象。我们实际上是创建一个 auto_ptr 类型的局部对象，该局部对象析构时，会将自身所拥有的指针空间释放，所以不会有内存泄漏；

3. auto_ptr 的构造函数是 explicit，阻止了一般指针隐式转换为 auto_ptr 的构造，所以不能直接将一般类型的指针赋值给 auto_ptr 类型的对象，必须用 auto_ptr 的构造函数创建对象；

4. 由于 auto_ptr 对象析构时会删除它所拥有的指针，所以使用时避免多个 auto_ptr 对象管理同一个指针；

5. Auto_ptr 内部实现，析构函数中删除对象用的是 delete 而不是 delete[]，所以 auto_ptr 不能管理数组；

6. auto_ptr 支持所拥有的指针类型之间的隐式类型转换。

7. 可以通过*和->运算符对 auto_ptr 所有用的指针进行提领操作；

8. T* get(),获得 auto_ptr 所拥有的指针；T* release()，释放 auto_ptr 的所有权，并将所有用的指针返回。

8、智能指针的循环引用

循环引用是指使用多个智能指针 share_ptr 时，出现了指针之间相互指向，从而形成环的情况，有点类似于死锁的情况，这种情况下，智能指针往往不能正常调用对象的析构函数，从而造成内存泄漏。举个例子：

```
Plaintext
#include <iostream>using namespace std;template <typename T>class
Node{public:        Node(const T&
value)                :_pPre(NULL)                ,
_pNext(NULL)                ,_value(value)
{               cout << "Node()" << endl;        }        ~Node()
```

```
{                cout << "~Node()" << endl;                cout <<
"this:" << this << endl;        }        shared_ptr<Node<T>>
_pPre;        shared_ptr<Node<T>> _pNext;        T _value;};void
Funtest(){        shared_ptr<Node<int>> sp1(new Node<int>(1));
shared_ptr<Node<int>> sp2(new Node<int>(2));        cout <<
"sp1.use_count:" << sp1.use_count() << endl;        cout <<
"sp2.use_count:" << sp2.use_count() << endl;        sp1->_pNext =
sp2; //sp1 的引用+1        sp2->_pPre = sp1; //sp2 的引用+1
cout << "sp1.use_count:" << sp1.use_count() << endl;        cout
<< "sp2.use_count:" << sp2.use_count() << endl;}int
main(){        Funtest();        system("pause");        return
0;}//输出结果
//Node()//Node()//sp1.use_count:1//sp2.use_count:1//sp1.use_count:
2//sp2.use_count:2
```

从上面 shared_ptr 的实现中我们知道了只有当引用计数减减之后等于 0，析构时才会释放对象，而上述情况造成了一个僵局，那就是析构对象时先析构 sp2,可是由于 sp2 的空间 sp1 还在使用中，所以 sp2.use_count 减减之后为 1，不释放，sp1 也是相同的道理，由于 sp1 的空间 sp2 还在使用中，所以 sp1.use_count 减减之后为 1，也不释放。sp1 等着 sp2 先释放，sp2 等着 sp1 先释放,二者互不相让，导致最终都没能释放，内存泄漏。

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。

9、手写实现智能指针类需要实现哪些函数?

1. 智能指针是一个数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录 SmartPointer<T*>对象的引用计数，一旦 T 类型对象的引用计数为 0，就释放该对象。

除了指针对象外，我们还需要一个引用计数的指针设定对象的值，并将引用计数计为 1，需要一个构造函数。新增对象还需要一个构造函数，析构函数负责引用计数减少和释放内存。

通过覆写赋值运算符，才能将一个旧的智能指针赋值给另一个指针，同时旧的引用计数减 1，新的引用计数加 1

1. 一个构造函数、拷贝构造函数、复制构造函数、析构函数、移动函数;

10、智能指针出现循环引用怎么解决?

弱指针用于专门解决 shared_ptr 循环引用的问题，weak_ptr 不会修改引用计数，即其

存在与否并不影响对象的引用计数器。循环引用就是：两个对象互相使用一个 shared_ptr 成员变量指向对方。弱引用并不对对象的内存进行管理，在功能上类似于普通指针，然而一个比较大的区别是，弱引用能检测到所管理的对象是否已经被释放，从而避免访问非法内存。

2.1.4、STL 模板库

1、什么是 STL?

C++ STL 从广义来讲包括了三类：算法，容器和迭代器。

- 算法包括排序，复制等常用算法，以及不同容器特定的算法。
- 容器就是数据的存放形式，包括序列式容器和关联式容器，序列式容器就是 list，vector 等，关联式容器就是 set，map 等。
- 迭代器就是在不暴露容器内部结构的情况下对容器的遍历。

2、解释一下什么是 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。

反之，用户自定义了析构函数，则称之为“non-trivial destructor”，这种析构函数**如果申请了新的空间一定要显式的释放，否则会造成内存泄露**

对于 trivial destructor，如果每次都进行调用，显然对效率是一种伤害，如何进行判断呢？

《STL 源码解析》中给出的说明是：

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

也就是说，在实际的应用当中，STL 库提供了相关的判断方法**__type_traits**，感兴趣的读者可以自行查阅使用方式。除了 trivial destructor，还有 trivial construct、trivial copy construct 等，如果能够对是否 trivial 进行区分，可以采用内存处理函数 memcpy()、malloc()等更加高效的完成相关操作，提升效率。

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。

因为 C++的语言机制保证了，当一个对象创建的时候，自动调用构造函数，当对象超

出作用域的时候会自动调用析构函数。所以，在 RAII 的指导下，我们应该使用类来管理资源，将资源和对象的生命周期绑定。

1. 智能指针（std::shared_ptr 和 std::unique_ptr）即 RAII 最具代表的实现，使用智能指针，可以实现自动的内存管理，再也不需要担心忘记 delete 造成的内存泄漏。

毫不夸张的来讲，有了智能指针，代码中几乎不需要再出现 delete 了。

4、迭代器：++it、it++哪个好，为什么

1. 前置返回一个引用，后置返回一个对象

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

1. 前置不会产生临时对象，后置必须产生临时对象，临时对象会导致效率降低

```
Plaintext
//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

C++11 正是通过引入右值引用来优化性能，具体来说是通过移动语义来避免无谓拷贝的问题，通过 move 语义来将临时生成的左值中的资源无代价的转移到另外一个对象中去，通过完美转发来解决不能按照参数实际类型来转发的问题（同时，完美转发获得的一个好处是可以实现移动语义）。

1. 在 C++11 中所有的值必属于左值、右值两者之一，右值又可以细分为纯右值、将亡值。在 C++11 中可以取地址的、有名字的就是左值，反之，不能取地址的、没有名字的就是右值（将亡值或纯右值）。举个例子，int a = b+c, a 就是左值，其有变量名为 a，通过&a 可以获取该变量的地址；表达式 b+c、函数 int func()的返回值是右值，在其被赋值给某一变量前，我们不能通过变量名找到它，＆(b+c)这样的操作则不会通过编译。

2. C++11 对 C++98 中的右值进行了扩充。在 C++11 中右值又分为纯右值（prvalue，Pure Rvalue）和将亡值（xvalue，eXpiring Value）。其中纯右值的概念等同于我们在 C++98 标准中右值的概念，指的是临时变量和不跟对象关联的字面量值；将亡值则是 C++11 新增的跟右值引用相关的表达式，这样表达式通常是将要被移动的对象（移为

他用），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。将亡值可以理解为通过“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。

3. 左值引用就是对一个左值进行引用的类型。右值引用就是对一个右值进行引用的类型，事实上，由于右值通常不具有名字，我们也只能通过引用的方式找到它的存在。右值引用和左值引用都是属于引用类型。无论是声明一个左值引用还是右值引用，都必须立即进行初始化。而其原因可以理解为是引用类型本身自己并不拥有所绑定对象的内存，只是该对象的一个别名。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。相对地，非常量左值只能接受非常量左值对其进行初始化。

4. 右值值引用通常不能绑定到任何的左值，要想绑定一个左值到右值引用，通常需要 std::move()将左值强制转换为右值。

左值和右值

左值：表示的是可以获取地址的表达式，它能出现在赋值语句的左边，对该表达式进行赋值。但是修饰符 const 的出现使得可以声明如下的标识符，它可以取得地址，但是没办法对其进行赋值

```
Plaintext
const int& a = 10;
```

右值：表示无法获取地址的对象，有常量值、函数返回值、lambda 表达式等。无法获取地址，但不表示其不可改变，当定义了右值的右值引用时就可以更改右值。

左值引用和右值引用

左值引用：传统的 C++中引用被称为左值引用

右值引用：C++11 中增加了右值引用，右值引用关联到右值时，右值被存储到特定位置，右值引用指向该特定位置，也就是说，右值虽然无法获取地址，但是右值引用是可以获取地址的，该地址表示临时对象的存储位置

这里主要说一下右值引用的特点：

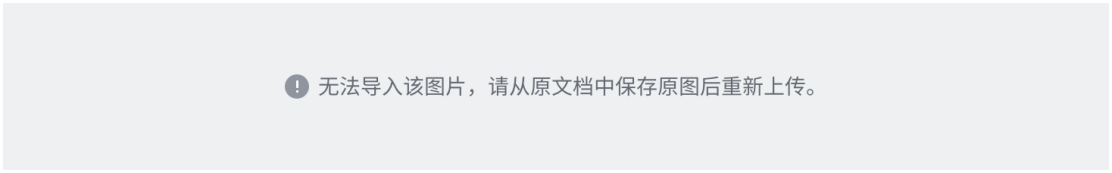
- 特点 1：通过右值引用的声明，右值又“重获新生”，其生命周期与右值引用类型变量的生命周期一样长，只要该变量还活着，该右值临时量将会一直存活下去
- 特点 2：右值引用独立于左值和右值。意思是右值引用类型的变量可能是左值也可能是右值
- 特点 3：T&& t 在发生自动类型推断的时候，它是左值还是右值取决于它的初始化。

举个例子:

```
C++
#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?

STL 中的 hashtable 使用的是**开链法**解决 hash 冲突问题，如下图所示。



hashtable 中的 bucket 所维护的 list 既不是 list 也不是 slist，而是其自己定义的由 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

在 hashtable 设计 bucket 的数量上，其内置了 28 个质数[53, 97, 193,...,429496729]，在创建 hashtable 时，会根据存入的元素个数选择大于等于元素个数的质数作为 hashtable 的容量（vector 的长度），其中每个 bucket 所维护的 linked-list 长度也等于 hashtable 的容量。如果插入 hashtable 的元素个数超过了 bucket 的容量，就要进行重建 table 操作，即找出下一个质数，创建新的 buckets vector，重新计算元素在新 hashtable 的位置。

7、简单说一下 STL 中的 traits 技法

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**，能够方面的让外界获取以下 5 中型别：

- 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，一般来说，type_traits 支持以下 5 中类型的判断：

```
Plaintext
__type_traits<T>::has_trivial_default_constructor__type_traits<T>::has_trivial_copy_constructor__type_traits<T>::has_trivial_assignment_operator__type_traits<T>::has_trivial_destructor__type_traits<T>::is_POD_type
```

由于编译器只针对 class object 形式的参数进行参数推到，因此上式的返回结果不应该是个 bool 值，实际上使用的是一种空的结构体：

```
Plaintext
struct __true_type{};struct __false_type{};
```

这两个结构体没有任何成员，不会带来其他的负担，又能满足需求，可谓一举两得

当然，如果我们自行定义了一个 Shape 类型，也可以针对这个 Shape 设计 type_traits 的特化版本

```
Plaintext
template<> struct __type_traits<Shape>{        typedef __true_type has_trivial_default_constructor;        typedef __false_type has_trivial_copy_constructor;        typedef __false_type has_trivial_assignment_operator;        typedef __false_type has_trivial_destructor;        typedef __false_type is_POD_type;};
```

8、STL 的两级空间配置器

1、首先明白为什么需要二级空间配置器?

我们知道动态开辟内存时，要在堆上申请，但若是我们需要

频繁的在堆开辟释放内存，则就会**在堆上造成很多外部碎片**，浪费了内存空间；

每次都要进行调用 **malloc、free** 函数等操作，使空间就会增加一些附加信息，降低了空间利用率；

随着外部碎片增多，内存分配器在找不到合适内存情况下需要合并空闲块，浪费了时间，大大降低了效率。

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。

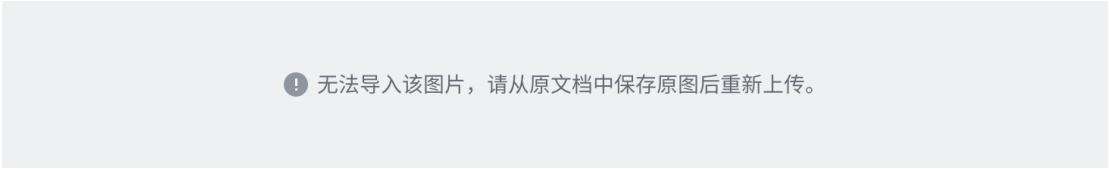
一级配置器

一级空间配置器中重要的函数就是 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 。大致过程是：

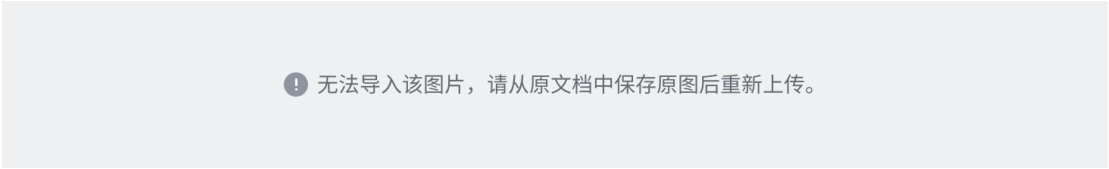
1、直接 allocate 分配内存，其实就是 malloc 来分配内存，成功则直接返回，失败就调用处理函数

2、如果用户自定义了内存分配失败的处理函数就调用，没有的话就返回异常

3、如果自定义了处理函数就进行处理，完事再继续分配试试



二级配置器



1、维护 16 条链表，分别是 0-15 号链表，最小 8 字节，以 8 字节逐渐递增，最大 128 字节，你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（如需要 13bytes 空间，我们会给它分配 16bytes 大小），在找到第 n 个链表后查看链表是否为空，如果不为空直接从对应的 free_list 中拔出，将已经拨出的指针向后移动一位。

2、对应的 free_list 为空，先看其内存池是不是空时，如果内存池不为空： （1）先检验它剩余空间是否够 20 个节点大小（即所需内存大小(提升后) * 20），若足够则直接从内存池中拿出 20 个节点大小空间，将其中一个分配给用户使用，另外 19 个当作自由链表中的区块挂在相应的 free_list 下，这样下次再有相同大小的内存需求时，可直

接拨出。（2）如果不够 20 个节点大小，则看它是否能满足 1 个节点大小，如果够的话则直接拿出一个分配给用户，然后从剩余的空间中分配尽可能多的节点挂在相应的 free_list 中。（3）如果连一个节点内存都不能满足的话，则将内存池中剩余的空间挂在相应的 free_list 中（找到相应的 free_list），然后再给内存池申请内存，转到 3。 3、内存池为空，申请内存 此时二级空间配置器会使用 malloc()从 heap 上申请内存，（一次所申请的内存大小为 2 * 所需节点内存大小（提升后）* 20 + 一段额外空间），申请 40 块，一半拿来用，一半放内存池中。 4、malloc 没有成功 在第三种情况下，如果 malloc()失败了，说明 heap 上没有足够空间分配给我们了，这时，二级空间配置器会从比所需节点空间大的 free_list 中一一搜索，从比它所需节点空间大的 free_list 中拔除一个节点来使用。如果这也没找到，说明比其大的 free_list 中都没有自由区块了，那就要调用一级适配器了。

释放时调用 deallocate()函数，若释放的 n>128，则调用一级空间配置器，否则就直接将内存块挂上自由链表的合适位置。

STL 二级空间配置器虽然解决了外部碎片与提高了效率，但它同时增加了一些缺点：

1.因为自由链表的管理问题，它会把我们需求的内存块自动提升为 8 的倍数，这时若你需要 1 个字节，它会给你 8 个字节，即浪费了 7 个字节，所以它又引入了内部碎片的问题，若相似情况出现很多次，就会造成很多内部碎片；

2.二级空间配置器是在堆上申请大块的狭义内存池，然后用自由链表管理，供现在使用，在程序执行过程中，它将申请的内存一块一块都挂在自由链表上，即不会还给操作系统，并且它的实现中所有成员全是静态的，所以它申请的所有内存只有在进程结束才会释放内存，还给操作系统，由此带来的问题有：1.即我不断的开辟小块内存，最后整个堆上的空间都被挂在自由链表上，若我想开辟大块内存就会失败；2.若自由链表上挂很多内存块没有被使用，当前进程又占着内存不释放，这时别的进程在堆上申请不到空间，也不可以使用当前进程的空闲内存，由此就会引发多种问题。

一级分配器

GC4.9 之后就没有第一级了，只有第二级

二级分配器

——default_alloc_template 剖析

有个自动调整的函数：你传入一个字节参数，表示你需要多大的内存，会自动帮你校对到第几号链表（0-15 号链表，最小 8 字节 最大 128 字节）

allocate 函数：如果要分配的内存大于 128 字节，就转用第一级分配器，否则也就是小于 128 字节。那么首先判断落在第几号链表，定位到了，先判断链表是不是空，如果是空就需要充值，（调节到 8 的倍数，默认一次申请 20 个区块，当然了也要判断 20 个是不是能够申请到，如果只申请到一个那就直接返回好了，不止一个的话，把第 2

到第 n 个挨个挂到当前链表上，第一个返回回去给容器用,n 是不大于 20 的，当然了如果不在 1-20 之间，那就是内存碎片了，那就先把碎片挂到某一条链表上，然后再重新 malloc 了，malloc 2*20 个块）去内存池去拿或者重新分配。不为空的话

9、 vector 与 list 的区别与应用？怎么找某 vector 或者 list 的倒数第二个元素

1. vector 数据结构 vector 和数组类似，拥有一段连续的内存空间，并且起始地址不变。因此能高效的进行随机存取，时间复杂度为 o(1);但因为内存空间是连续的，所以在进行插入和删除操作时，会造成内存块的拷贝，时间复杂度为 o(n)。另外，当数组中内存空间不够时，会重新申请一块内存空间并进行内存拷贝。连续存储结构：vector 是可以实现动态增长的对象数组，支持对数组高效率的访问和在数组尾端的删除和插入操作，在中间和头部删除和插入相对不易，需要挪动大量的数据。它与数组最大的区别就是 vector 不需程序员自己去考虑容量问题，库里面本身已经实现了容量的动态增长，而数组需要程序员手动写入扩容函数进形扩容。

2. list 数据结构 list 是由双向链表实现的，因此内存空间是不连续的。只能通过指针访问数据，所以 list 的随机存取非常没有效率，时间复杂度为 o(n);但由于链表的特点，能高效地进行插入和删除。非连续存储结构：list 是一个双链表结构，支持对链表的双向遍历。每个节点包括三个信息：元素本身，指向前一个元素的节点（prev）和指向下一个元素的节点（next）。因此 list 可以高效率的对数据元素任意位置进行访问和插入删除等操作。由于涉及对额外指针的维护，所以开销比较大。

区别：

vector 的随机访问效率高，但在插入和删除时（不包括尾部）需要挪动数据，不易操作。list 的访问要遍历整个链表，它的随机访问效率低。但对数据的插入和删除操作等都比较方便，改变指针的指向即可。list 是单向的，vector 是双向的。vector 中的迭代器在使用后就失效了，而 list 的迭代器在使用之后还可以继续使用。

1.

int mySize = vec.size();vec.at(mySize -2);

list 不提供随机访问，所以不能用下标直接访问到某个位置的元素，要访问 list 里的元素只能遍历，不过你要是只需要访问 list 的最后 N 个元素的话，可以用反向迭代器来遍历：

10、STL 中 vector 删除其中的元素，迭代器如何变化？为什么是两倍扩容？释放空间？

size()函数返回的是已用空间大小，capacity()返回的是总空间大小，capacity()-size()则是剩余的可用空间大小。当 size()和 capacity()相等，说明 vector 目前的空间已被用

完，如果再添加新元素，则会引起 vector 空间的动态增长。

由于动态增长会引起重新分配内存空间、拷贝原空间、释放原空间，这些过程会降低程序效率。因此，可以使用 reserve(n)预先分配一块较大的指定大小的内存空间，这样当指定大小的内存空间未使用完时，是不会重新分配内存空间的，这样便提升了效率。只有当 n>capacity()时，调用 reserve(n)才会改变 vector 容量。

resize()成员函数只改变元素的数目，不改变 vector 的容量。

1、空的 vector 对象，size()和 capacity()都为 0

2、当空间大小不足时，新分配的空间大小为原空间大小的 2 倍。

3、使用 reserve()预先分配一块内存后，在空间未满的情况下，不会引起重新分配，从而提升了效率。

4、当 reserve()分配的空间比原空间小时，是不会引起重新分配的。

5、resize()函数只改变容器的元素数目，未改变容器大小。

6、用 reserve(size_type)只是扩大 capacity 值，这些内存空间可能还是“野”的，如果此时使用“[ ]”来访问，则可能会越界。而 resize(size_type new_size)会真正使容器具有 new_size 个对象。

不同的编译器，vector 有不同的扩容大小。在 vs 下是 1.5 倍，在 GCC 下是 2 倍;

空间和时间的权衡。简单来说， 空间分配的多，平摊时间复杂度低，但浪费空间也多。

使用 k=2 增长因子的问题在于，每次扩展的新尺寸必然刚好大于之前分配的总和，也就是说，之前分配的内存空间不可能被使用。这样对内存不友好。最好把增长因子设为(1,2)

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。

11、Vector 如何释放空间?

由于 vector 的内存占用空间只增不减，比如你首先分配了 10,000 个字节，然后 erase 掉后面 9,999 个，留下一个有效元素，但是内存占用仍为 10,000 个。所有内存空间是在 vector 析构时候才能被系统回收。empty()用来检测容器是否为空的，clear()可以清空所有元素。但是即使 clear()，vector 所占用的内存空间依然如故，无法保证内存的回收。

如果需要空间动态缩小，可以考虑使用 deque。如果 vector，可以用 swap()来帮助你释放内存。

```
Plaintext
vector(Vec).swap(Vec); 将 Vec 的内存空洞清除； vector().swap(Vec); 清
```

空 Vec 的内存；

12、容器内部删除一个元素

1. 顺序容器（序列式容器，比如 vector、deque）

erase 迭代器不仅使所指向被删除的迭代器失效，而且使被删元素之后的所有迭代器失效(list 除外)，所以不能使用 erase(it++)的方式，但是 erase 的返回值是下一个有效迭代器；

It = c.erase(it);

1. 关联容器(关联式容器，比如 map、set、multimap、multiset 等)

erase 迭代器只是被删除元素的迭代器失效，但是返回值是 void，所以要采用 erase(it++)的方式删除迭代器；

c.erase(it++)

13、STL 迭代器如何实现

1、 迭代器是一种抽象的设计理念，通过迭代器可以在不了解容器内部原理的情况下遍历容器，除此之外，STL 中迭代器一个最重要的作用就是作为容器与 STL 算法的粘合剂。

2、 迭代器的作用就是提供一个遍历容器内部所有元素的接口，因此迭代器内部必须保存一个与容器相关联的指针，然后重载各种运算操作来遍历，其中最重要的是*运算符与->运算符，以及++、--等可能需要重载的运算符重载。这和 C++中的智能指针很像，智能指针也是将一个指针封装，然后通过引用计数或是其他方法完成自动释放内存的功能。

3、最常用的迭代器的相应型别有五种：value type、difference type、pointer、reference、iterator catagoly;

14、map、set 是怎么实现的，红黑树是怎么能够同时实现这两种容器？ 为什么使用红黑树?

1. 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；

2. 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

3. 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。

15、如何在共享内存上使用 STL 标准库?

1. 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。

我们没必要再为共享内存设计其他额外的数据结构，另外，STL 的高度可扩展性将为 IPC 所驱使。STL 容器被良好的封装，默认情况下有它们自己的内存管理方案。

当一个元素被插入到一个 STL 列表(list)中时，列表容器自动为其分配内存，保存数据。考虑到要将 STL 容器放到共享内存中，而容器却自己在堆上分配内存。

一个最笨拙的办法是在堆上构造 STL 容器，然后把容器复制到共享内存，并且确保所有容器的内部分配的内存指向共享内存中的相应区域，这基本是个不可能完成的任务。

1. 假设进程 A 在共享内存中放入了数个容器，进程 B 如何找到这些容器呢?

一个方法就是进程 A 把容器放在共享内存中的确定地址上（fixed offsets），则进程 B 可以从该已知地址上获取容器。另外一个改进点的办法是，进程 A 先在共享内存某块确定地址上放置一个 map 容器，然后进程 A 再创建其他容器，然后给其取个名字和地址一并保存到这个 map 容器里。

进程 B 知道如何获取该保存了地址映射的 map 容器，然后同样再根据名字取得其他容器的地址。

16、map 插入方式有哪几种?

```
Plaintext
1) 用 insert 函数插入 pair 数据，mapStudent.insert(pair<int, string>(1, "student_one")); 2) 用 insert 函数插入 value_type 数据 mapStudent.insert(map<int, string>::value_type (1, "student_one"));3) 在 insert 函数中使用 make_pair()函数 mapStudent.insert(make_pair(1, "student_one")); 4) 用数组方式插入数据 mapStudent[1] = "student_one";
```

17、STL 中 unordered_map(hash_map)和 map 的区别，hash_map 如何解决冲突以及扩容

1. unordered_map 和 map 类似，都是存储的 key-value 的值，可以通过 key 快速索引到 value。不同的是 unordered_map 不会根据 key 的大小进行排序，

2. 存储时是根据 key 的 hash 值判断元素是否相同，即 unordered_map 内部元素是无序的，而 map 中的元素是按照二叉搜索树存储，进行中序遍历会得到有序遍历。

3. 所以使用时 map 的 key 需要定义 operator<。而 unordered_map 需要定义 hash_value 函数并且重载 operator==。但是很多系统内置的数据类型都自带这些，

4. 那么如果是自定义类型，那么就需要自己重载 operator<或者 hash_value()了。

5. 如果需要内部元素自动排序，使用 map，不需要排序使用 unordered_map

6. unordered_map 的底层实现是 hash_table;

7. hash_map 底层使用的是 hash_table，而 hash_table 使用的开链法进行冲突避免，所有 hash_map 采用开链法进行冲突解决。

8. **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。

9. **扩容(resize)**就是重新计算容量，向 HashMap 对象里不停的添加元素，而 HashMap 对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。

18、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？

1. 通过下标访问 vector 中的元素时不会做边界检查，即便下标越界。

也就是说，下标与 first 迭代器相加的结果超过了 finish 迭代器的位置，程序也不会报错，而是返回这个地址中存储的值。

如果想在访问 vector 中的元素时首先进行边界检查，可以使用 vector 中的 at 函数。通过使用 at 函数不但可以通过下标访问 vector 中的元素，而且在 at 函数内部会对下标进行边界检查。

1. map 的下标运算符[]的作用是：将 key 作为下标去执行查找，并返回相应的值；如果不存在这个 key，就将一个具有该 key 和 value 的某人值插入这个 map。

2. erase()函数，只能删除内容，不能改变容量大小;

erase 成员函数，它删除了 itVect 迭代器指向的元素，并且返回要被删除的 itVect 之后的迭代器，迭代器相当于一个智能指针;clear()函数，只能清空内容，不能改变容量大小;如果要想在删除内容的同时释放内存，那么你可以选择 deque 容器。

19、map 中[]与 find 的区别？

1. 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。

2. 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。

20、 STL 中 list 与 queue 之间的区别

1. list 不再能够像 vector 一样以普通指针作为迭代器，因为其节点不保证在存储空间

中连续存在；

2. list 插入操作和结合才做都不会造成原有的 list 迭代器失效;

3. list 不仅是一个双向链表，而且还是一个环状双向链表，所以它只需要一个指针；

4. list 不像 vector 那样有可能在空间不足时做重新配置、数据移动的操作，所以插入前的所有迭代器在插入操作之后都仍然有效；

5. deque 是一种双向开口的连续线性空间，所谓双向开口，意思是可以在头尾两端分别做元素的插入和删除操作；可以在头尾两端分别做元素的插入和删除操作；

6. deque 和 vector 最大的差异，一在于 deque 允许常数时间内对起头端进行元素的插入或移除操作，二在于 deque 没有所谓容量概念，因为它是动态地以分段连续空间组合而成，随时可以增加一段新的空间并链接起来，deque 没有所谓的空间保留功能。

21、STL 中的 allocator、deallocator

1. 第一级配置器直接使用 malloc()、free()和 relloc()，第二级配置器视情况采用不同的策略：当配置区块超过 128bytes 时，视之为足够大，便调用第一级配置器；当配置器区块小于 128bytes 时，为了降低额外负担，使用复杂的内存池整理方式，而不再用一级配置器；

2. 第二级配置器主动将任何小额区块的内存需求量上调至 8 的倍数，并维护 16 个 free-list，各自管理大小为 8~128bytes 的小额区块；

3. 空间配置函数 allocate()，首先判断区块大小，大于 128 就直接调用第一级配置器，小于 128 时就检查对应的 free-list。如果 free-list 之内有可用区块，就直接拿来用，如果没有可用区块，就将区块大小调整至 8 的倍数，然后调用 refill()，为 free-list 重新分配空间；

4. 空间释放函数 deallocate()，该函数首先判断区块大小，大于 128bytes 时，直接调用一级配置器，小于 128bytes 就找到对应的 free-list 然后释放内存。

22、STL 中 hash_map 扩容发生什么?

1. hash table 表格内的元素称为桶（bucket),而由桶所链接的元素称为节点（node),其中存入桶元素的容器为 stl 本身很重要的一种序列式容器——vector 容器。之所以选择 vector 为存放桶元素的基础容器，主要是因为 vector 容器本身具有动态扩容能力，无需人工干预。

2. 向前操作：首先尝试从目前所指的节点出发，前进一个位置（节点），由于节点被安置于 list 内，所以利用节点的 next 指针即可轻易完成前进操作，如果目前正巧是 list 的尾端，就跳至下一个 bucket 身上，那正是指向下一个 list 的头部节点。

23、常见容器性质总结?

1.vector 底层数据结构为数组， 支持快速随机访问

2.list 底层数据结构为双向链表，支持快速增删

3.deque 底层数据结构为一个中央控制器和多个缓冲区，详细见 STL 源码剖析 P146，支持首尾（中间不能）快速增删，也支持随机访问

deque 是一个双端队列(double-ended queue)，也是在堆中保存内容的.它的保存形式如下:

[堆 1] --> [堆 2] -->[堆 3] --> ...

每个堆保存好几个元素,然后堆和堆之间有指针指向,看起来像是 list 和 vector 的结合品.

4.stack 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时

5.queue 底层一般用 list 或 deque 实现，封闭头部即可，不用 vector 的原因应该是容量大小有限制，扩容耗时（stack 和 queue 其实是适配器,而不叫容器，因为是对容器的再封装）

6.priority_queue 的底层数据结构一般为 vector 为底层容器，堆 heap 为处理规则来管理底层容器实现

7.set 底层数据结构为红黑树，有序，不重复

8.multiset 底层数据结构为红黑树，有序，可重复

9.map 底层数据结构为红黑树，有序，不重复

10.multimap 底层数据结构为红黑树，有序，可重复

11.unordered_set 底层数据结构为 hash 表，无序，不重复

12.unordered_multiset 底层数据结构为 hash 表，无序，可重复

13.unordered_map 底层数据结构为 hash 表，无序，不重复

14.unordered_multimap 底层数据结构为 hash 表，无序，可重复

24、vector 的增加删除都是怎么做的？为什么是 1.5 或者是 2 倍?

1. 新增元素：vector 通过一个连续的数组存放元素，如果集合已满，在新增数据的时候，就要分配一块更大的内存，将原来的数据复制过来，释放之前的内存，在插入新增的元素；

2. 对 vector 的任何操作，一旦引起空间重新配置，指向原 vector 的所有迭代器就都失效了；

3. 初始时刻 vector 的 capacity 为 0，塞入第一个元素后 capacity 增加为 1；

4. 不同的编译器实现的扩容方式不一样，VS2015 中以 1.5 倍扩容，GCC 以 2 倍扩

容。

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。

1. 考虑可能产生的堆空间浪费，成倍增长倍数不能太大，使用较为广泛的扩容方式有两种，以 2 二倍的方式扩容，或者以 1.5 倍的方式扩容。

2. 以 2 倍的方式扩容，导致下一次申请的内存必然大于之前分配内存的总和，导致之前分配的内存不能再被使用，所以最好倍增长因子设置为(1,2)之间：

3. 向量容器 vector 的成员函数 pop_back()可以删除最后一个元素.

4. 而函数 erase()可以删除由一个 iterator 指出的元素，也可以删除一个指定范围的元素。

5. 还可以采用通用算法 remove()来删除 vector 容器中的元素.

6. 不同的是：采用 remove 一般情况下不会改变容器的大小，而 pop_back()与 erase()等成员函数会改变容器的大小。

25、说一下 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

26、STL 中迭代器失效的情况有哪些?

以 vector 为例：

插入元素：

1、尾后插入：size < capacity 时，首迭代器不失效尾迭代失效（未重新分配空间），size == capacity 时，所有迭代器均失效（需要重新分配空间）。

2、中间插入：中间插入：size < capacity 时，首迭代器不失效但插入元素之后所有迭代器失效，size == capacity 时，所有迭代器均失效。

删除元素：

尾后删除：只有尾迭代失效。

中间删除：删除位置之后所有迭代失效。

deque 和 vector 的情况类似,

而 list 双向链表每一个节点内存不连续, 删除节点仅当前迭代器失效,erase 返回下一个有效迭代器;

map/set 等关联容器底层是红黑树删除节点不会影响其他节点的迭代器, 使用递增方法获取下一个迭代器 mmp.erase(iter++);

unordered_(hash) 迭代器意义不大, rehash 之后, 迭代器应该也是全部失效.

27、STL 中 vector 的实现

vector 是一种序列式容器，其数据安排以及操作方式与 array 非常类似，两者的唯一差别就是对于空间运用的灵活性，众所周知，array 占用的是静态空间，一旦配置了就不可以改变大小，如果遇到空间不足的情况还要自行创建更大的空间，并手动将数据拷贝到新的空间中，再把原来的空间释放。vector 则使用灵活的动态空间配置，维护一块**连续的线性空间**，在空间不足时，可以自动扩展空间容纳新元素，做到按需供给。其在扩充空间的过程中仍然需要经历：**重新配置空间，移动数据，释放原空间**等操作。这里需要说明一下动态扩容的规则：以原大小的两倍配置另外一块较大的空间（或者旧长度+新增元素的个数），源码：

```
Plaintext
const size_type len  = old_size + max(old_size, n);
```

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

测试代码：

```
C++
#include <iostream>#include <vector>using namespace std;int
main(){    //在 Linux + GCC 下        vector<int> res(2,0);
cout << res.capacity() <<endl; //2        res.push_back(1);
cout << res.capacity() <<endl;//4        res.push_back(2);
res.push_back(3);    cout << res.capacity() <<endl;//8
return 0;            //在 win 10 + VS2019 下    vector<int>
res(2,0);        cout << res.capacity() <<endl; //2
res.push_back(1);        cout << res.capacity() <<endl;//3
res.push_back(2);        res.push_back(3);    cout <<
res.capacity() <<endl;//6     }
```

运行上述代码，一开始配置了一块长度为 2 的空间，接下来插入一个数据，长度变为原来的两倍，为 4，此时已占用的长度为 3，再继续两个数据，此时长度变为 8，可以

清晰的看到空间的变化过程

需要注意的是，频繁对 vector 调用 push_back()对性能是有影响的，这是因为每插入一个元素，如果空间够用的话还能直接插入，若空间不够用，则需要重新配置空间，移动数据，释放原空间等操作，对程序性能会造成一定的影响

28、STL 中 slist 的实现

list 是双向链表，而 slist（single linked list）是单向链表，它们的主要区别在于：前者的迭代器是双向的 Bidirectional iterator，后者的迭代器属于单向的 Forward iterator。虽然 slist 的很多功能不如 list 灵活，但是其所耗用的空间更小，操作更快。

根据 STL 的习惯，插入操作会将新元素插入到指定位置之前，而非之后，然而 slist 是不能回头的，只能往后走，因此在 slist 的其他位置插入或者移除元素是十分不明智的，但是在 slist 开头却是可取的，slist 特别提供了 insert_after()和 erase_after 供灵活应用。考虑到效率问题，slist 只提供 push_front()操作，元素插入到 slist 后，存储的次序和输入的次序是相反的

slist 的单向迭代器如下图所示：

slist 默认采用 alloc 空间配置器配置节点的空间，其数据结构主要代码如下

```
Plaintext
template <class T, class Allco = alloc>class
slist{        ...private:    ...    static list_node*
create_node(const value_type& x){}//配置空间、构造元素    static
void destroy_node(list_node* node){}//析构函数、释放空间 private:
list_node_base head; //头部 public:    iterator begin(){}
iterator end(){}    size_type size(){}    bool empty(){}    void
swap(slist& L){}//交换两个 slist，只需要换 head 即可    reference
front(){} //取头部元素    void push_front(const value& x){}//头部插
入元素    void pop_front(){}//从头部取走元素    ...}
```

举个例子：

```
Plaintext
#include <forward_list>#include <algorithm>#include
<iostream>using namespace std;int main(){        forward_list<int>
fl;        fl.push_front(1);        fl.push_front(3);
fl.push_front(2);        fl.push_front(6);
```

```
fl.push_front(5);        forward_list<int>::iterator ite1 =
fl.begin();        forward_list<int>::iterator ite2 = fl.end();
for(;ite1 != ite2; ++ite1)        {                cout << *ite1
<<" "; // 5 6 2 3 1        }        cout << endl;        ite1 =
find(fl.begin(), fl.end(), 2); //寻找 2 的位置        if (ite1 !=
ite2)                fl.insert_after(ite1, 99);        for (auto
it : fl)        {                cout << it << " ";  //5 6 2 99 3
1        }        cout << endl;        ite1 = find(fl.begin(),
fl.end(), 6); //寻找 6 的位置        if (ite1 != ite2)
fl.erase_after(ite1);        for (auto it : fl)
{                cout << it << " ";  //5 6 99 3 1        }
cout << endl;                return 0;}
```

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。

29、STL 中 list 的实现

相比于 vector 的连续线型空间，list 显得复杂许多，但是它的好处在于插入或删除都只作用于一个元素空间，因此 list 对空间的运用是十分精准的，对任何位置元素的插入和删除都是常数时间。list 不能保证节点在存储空间中连续存储，也拥有迭代器，迭代器的“++”、“--”操作对于的是指针的操作，list 提供的迭代器类型是双向迭代器：Bidirectional iterators。

list 节点的结构见如下源码：

```
Plaintext
template <class T>struct __list_node{    typedef void*
void_pointer;    void_pointer prev;    void_pointer next;    T
data;}
```

从源码可看出 list 显然是一个双向链表。list 与 vector 的另一个区别是，在插入和接合操作之后，都不会造成原迭代器失效，而 vector 可能因为空间重新配置导致迭代器失效。

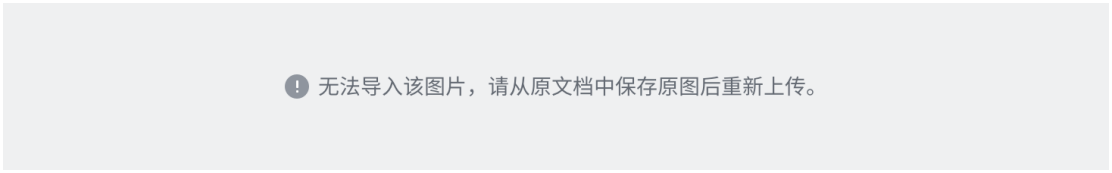
此外 list 也是一个环形链表，因此只要一个指针便能完整表现整个链表。list 中 node 节点指针始终指向尾端的一个空白节点，因此是一种“前闭后开”的区间结构

list 的空间管理默认采用 alloc 作为空间配置器，为了方便的以节点大小为配置单位，还定义一个 list_node_allocator 函数可一次性配置多个节点空间

由于 list 的双向特性，其支持在头部（front)和尾部（back)两个方向进行 push 和 pop 操作，当然还支持 erase，splice，sort，merge，reverse，sort 等操作，这里不再详细阐述。

30、STL 中的 deque 的实现

vector 是单向开口（尾部）的连续线性空间，deque 则是一种双向开口的连续线性空间，虽然 vector 也可以在头尾进行元素操作，但是其头部操作的效率十分低下（主要是涉及到整体的移动）



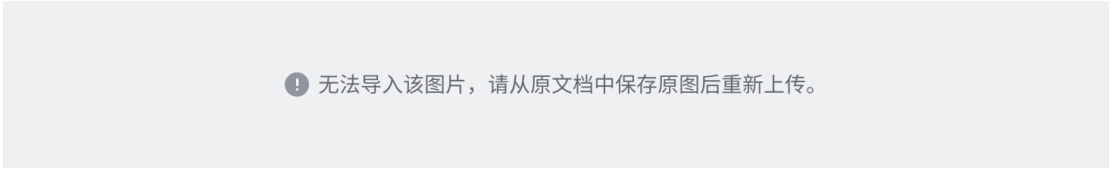
deque 和 vector 的最大差异一个是 deque 运行在常数时间内对头端进行元素操作，二是 deque 没有容量的概念，它是动态地以分段连续空间组合而成，可以随时增加一段新的空间并链接起来

deque 虽然也提供随机访问的迭代器，但是其迭代器并不是普通的指针，其复杂程度比 vector 高很多，因此除非必要，否则一般使用 vector 而非 deque。如果需要对 deque 排序，可以先将 deque 中的元素复制到 vector 中，利用 sort 对 vector 排序，再将结果复制回 deque

deque 由一段一段的定量连续空间组成，一旦需要增加新的空间，只要配置一段定量连续空间拼接在头部或尾部即可，因此 deque 的最大任务是如何维护这个整体的连续性

deque 的数据结构如下：

```
Plaintext
class deque{    ...protected:    typedef pointer* map_pointer;//指向 map 指针的指针    map_pointer map;//指向 map    size_type map_size;//map 的大小 public:    ...    iterator begin(); itertator end();    ...}
```



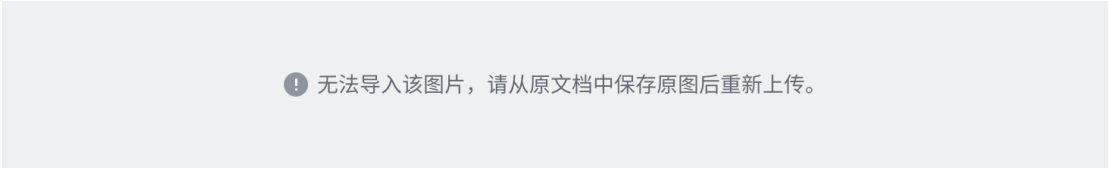
deque 内部有一个指针指向 map，map 是一小块连续空间，其中的每个元素称为一个节点，node，每个 node 都是一个指针，指向另一段较大的连续空间，称为缓冲区，这里就是 deque 中实际存放数据的区域，默认大小 512bytes。整体结构如上图所示。

deque 的迭代器数据结构如下：

```
Plaintext
struct __deque_iterator{    ...    T* cur;//迭代器所指缓冲区当前的元
```

```
素    T* first;//迭代器所指缓冲区第一个元素    T* last;//迭代器所指缓冲区最后一个元素    map_pointer node;//指向 map 中的 node    ...}
```

从 deque 的迭代器数据结构可以看出，为了保持与容器联结，迭代器主要包含上述 4 个元素

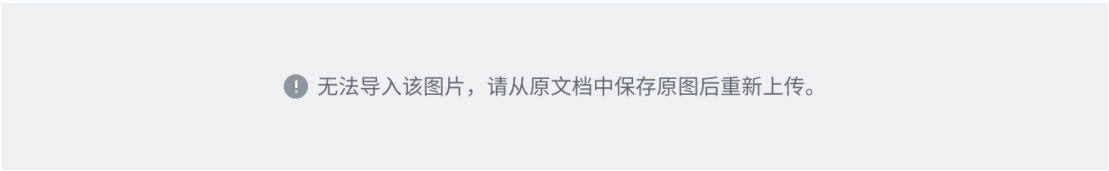


deque 迭代器的“++”、“--”操作是远比 vector 迭代器繁琐，其主要工作在于缓冲区边界，如何从当前缓冲区跳到另一个缓冲区，当然 deque 内部在插入元素时，如果 map 中 node 数量全部使用完，且 node 指向的缓冲区也没有多余的空间，这时会配置新的 map（2 倍于当前+2 的数量）来容纳更多的 node，也就是可以指向更多的缓冲区。在 deque 删除元素时，也提供了元素的析构和空闲缓冲区空间的释放等机制。

31、STL 中 stack 和 queue 的实现

stack

stack（栈）是一种先进后出（First In Last Out）的数据结构，只有一个入口和出口，那就是栈顶，除了获取栈顶元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：



stack 这种单向开口的数据结构很容易由**双向开口的 deque 和 list** 形成，只需要根据 stack 的性质对应移除某些接口即可实现，stack 的源码如下：

```
Plaintext
template <class T, class Sequence = deque<T> >class stack{        ...protected:    Sequence c;public:    bool empty(){return c.empty();}    size_type size() const{return c.size();}    reference top() const {return c.back();}    const_reference top() const{return c.back();}    void push(const value_type& x){c.push_back(x);}    void pop(){c.pop_back();}};
```

从 stack 的数据结构可以看出，其所有操作都是围绕 Sequence 完成，而 Sequence 默认是 deque 数据结构。stack 这种“修改某种接口，形成另一种风貌”的行为，成为 adapter(配接器)。常将其归类为 container adapter 而非 container

stack 除了默认使用 deque 作为其底层容器之外，也可以使用双向开口的 list，只需要

在初始化 stack 时，将 list 作为第二个参数即可。由于 stack 只能操作顶端的元素，因此其内部元素无法被访问，也不提供迭代器。

queue

queue（队列）是一种先进先出（First In First Out）的数据结构，只有一个入口和一个出口，分别位于最底端和最顶端，出口元素外，没有其他方法可以获取到内部的其他元素，其结构图如下：

类似的，queue 这种“先进先出”的数据结构很容易由双向开口的 deque 和 list 形成，只需要根据 queue 的性质对应移除某些接口即可实现，queue 的源码如下：

```
Plaintext
template <class T, class Sequence = deque<T> >class
queue{        ...protected:    Sequence c;public:    bool
empty(){return c.empty();}    size_type size() const{return
c.size();}    reference front() const {return c.front();}
const_reference front() const{return c.front();}    void
push(const value_type& x){c.push_back(x);}    void
pop(){c.pop_front();}};
```

从 queue 的数据结构可以看出，其所有操作都也都是是围绕 Sequence 完成，Sequence 默认也是 deque 数据结构。queue 也是一类 container adapter。

同样，queue 也可以使用 list 作为底层容器，不具有遍历功能，没有迭代器。

32、STL 中的 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，如下图所示就是一颗完全二叉树

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。

因此我们可以使用一个 array 和一组 heap 算法来实现 max heap（每个节点的值大于等于其子节点的值）和 min heap（每个节点的值小于等于其子节点的值）。由于 array 不能动态的改变空间大小，用 vector 代替 array 是一个不错的选择。

那 heap 算法有哪些？常见有的插入、弹出、排序和构造算法，下面一一进行描述。

push_heap 插入算法

由于完全二叉树的性质，新插入的元素一定是位于树的最底层作为叶子节点，并填补由左至右的第一个空格。事实上，在刚执行插入操作时，新元素位于底层 vector 的 end()处，之后是一个称为 percolate up（上溯）的过程，举个例子如下图：

新元素 50 在插入堆中后，先放在 vector 的 end()存着，之后执行上溯过程，调整其根结点的位置，以便满足 max heap 的性质，如果了解大根堆的话，这个原理跟大根堆的调整过程是一样的。

pop_heap 算法

heap 的 pop 操作实际弹出的是根节点吗，但在 heap 内部执行 pop_heap 时，只是将其移动到 vector 的最后位置，然后再为这个被挤走的元素找到一个合适的安放位置，使整颗树满足完全二叉树的条件。这个被挤掉的元素首先会与根结点的两个子节点比较，并与较大的子节点更换位置，如此一直往下，直到这个被挤掉的元素大于左右两个子节点，或者下放到叶节点为止，这个过程称为 percolate down（下溯）。举个例子：

根节点 68 被 pop 之后，移到了 vector 的最底部，将 24 挤出，24 被迫从根节点开始与其子节点进行比较，直到找到合适的位置安身，需要注意的是 pop 之后元素并没有被移走，如果要将其移走，可以使用 pop_back()。

sort 算法

一言以蔽之，因为 pop_heap 可以将当前 heap 中的最大值置于底层容器 vector 的末尾，heap 范围减 1，那么不断的执行 pop_heap 直到树为空，即可得到一个递增序列。

make_heap 算法

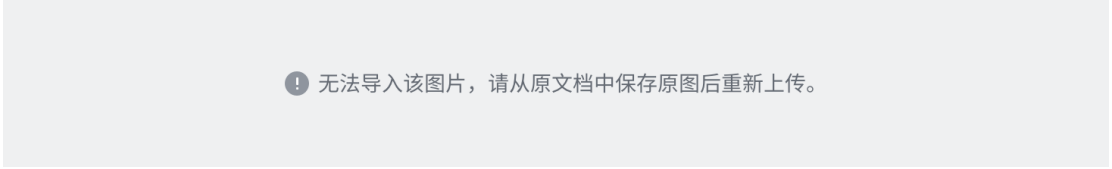
将一段数据转化为 heap，一个一个数据插入，调用上面说的两种 percolate 算法即可。

代码实测：

```
Plaintext
#include <iostream>#include <algorithm>#include <vector>using
namespace std;int main(){        vector<int> v =
{ 0,1,2,3,4,5,6 };        make_heap(v.begin(), v.end()); //以
vector 为底层容器        for (auto i : v)
{                cout << i << " "; // 6 4 5 3 1 0 2        }
cout << endl;        v.push_back(7);        push_heap(v.begin(),
v.end());        for (auto i : v)        {                cout <<
i << " "; // 7 6 5 4 1 0 2 3        }        cout << endl;
pop_heap(v.begin(), v.end());        cout << v.back() << endl; //
7        v.pop_back();        for (auto i : v)
{                cout << i << " "; // 6 4 5 3 1 0 2        }
cout << endl;        sort_heap(v.begin(), v.end());        for
(auto i : v)        {                cout << i << " "; // 0 1 2 3
4 5 6        }        return 0;}
```

33、STL 中的 priority_queue 的实现

priority_queue，优先队列，是一个拥有权值观念的 queue，它跟 queue 一样是顶部入口，底部出口，在插入元素时，元素并非按照插入次序排列，它会自动根据权值（通常是元素的实值）排列，权值最高，排在最前面，如下图所示。



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。关键的源码如下：

```
Plaintext
template <class T, class Squence = vector<T>, class Compare =
less<typename Sequence::value_tyoe> >class
priority_queue{        ...protected:    Sequence c; // 底层容器
Compare comp; // 元素大小比较标准 public:    bool empty() const
{return c.empty();}    size_type size() const {return c.size();}
const_reference top() const {return c.front()}    void push(const
value_type& x)    {        c.push_heap(x);
push_heap(c.begin(), c.end(),comp);    }    void pop()
{        pop_heap(c.begin(), c.end(),comp);
c.pop_back();    }};
```

priority_queue 的所有元素，进出都有一定的规则，只有 queue 顶端的元素（权值最高者），才有机会被外界取用，它没有遍历功能，也不提供迭代器

举个例子：

```
Plaintext
#include <queue>#include <iostream>using namespace std;int
main(){        int ia[9] = {0,4,1,2,3,6,5,8,7 };
priority_queue<int> pq(ia, ia + 9);        cout << pq.size()
<<endl;  // 9        for(int i = 0; i < pq.size(); i++)
{               cout << pq.top() << " "; // 8 8 8 8 8 8 8 8
8        }        cout << endl;        while (!pq.empty())
{               cout << pq.top() << ' ';// 8 7 6 5 4 3 2 1 0
pq.pop();        }        return 0;}
```

34、STL 中 set 的实现?

STL 中的容器可分为序列式容器（sequence）和关联式容器（associative），set 属于关联式容器。

set 的特性是，所有元素都会根据元素的值自动被排序（默认升序），set 元素的键值就是实值，实值就是键值，set 不允许有两个相同的键值

set 不允许迭代器修改元素的值，其迭代器是一种 constance iterators

标准的 STL set 以 RB-tree（红黑树）作为底层机制，几乎所有的 set 操作行为都是转调用 RB-tree 的操作行为，这里补充一下红黑树的特性：

- 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

关于红黑树的具体操作过程，比较复杂读者可以翻阅《算法导论》详细了解。

举个例子：

```
Plaintext
#include <set>#include <iostream>using namespace std;int
main(){        int i;        int ia[5] = { 1,2,3,4,5 };
set<int> s(ia, ia + 5);        cout << s.size() << endl; // 5
cout << s.count(3) << endl; // 1        cout << s.count(10) <<
endl; // 0        s.insert(3); //再插入一个 3        cout <<
s.size() << endl; // 5        cout << s.count(3) << endl; // 1
s.erase(1);        cout << s.size() << endl; // 4
```

```
set<int>::iterator b = s.begin();        set<int>::iterator e =
s.end();        for (; b != e; ++b)                cout << *b << "
"; // 2 3 4 5        cout << endl;        b = find(s.begin(),
s.end(), 5);        if (b != s.end())                cout << "5
found" << endl; // 5 found        b = s.find(2);        if (b !=
s.end())                cout << "2 found" << endl; // 2 found
b = s.find(1);        if (b == s.end())                cout << "1
not found" << endl; // 1 not found        return 0;}
```

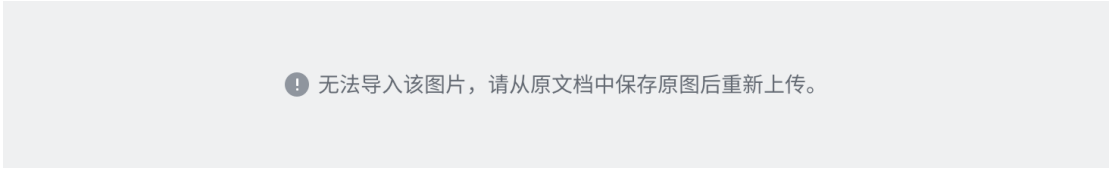
关联式容器尽量使用其自身提供的 find()函数查找指定的元素，效率更高，因为 STL 提供的 find()函数是一种顺序搜索算法。

35、STL 中 map 的实现

map 的特性是所有元素会根据键值进行自动排序。map 中所有的元素都是 pair，拥有键值(key)和实值(value)两个部分，并且不允许元素有相同的 key

一旦 map 的 key 确定了，那么是无法修改的，但是可以修改这个 key 对应的 value，因此 map 的迭代器既不是 constant iterator，也不是 mutable iterator

标准 STL map 的底层机制是 RB-tree（红黑树），另一种以 hash table 为底层机制实现的称为 hash_map。map 的架构如下图所示



map 的在构造时缺省采用递增排序 key，也使用 alloc 配置器配置空间大小，需要注意的是在插入元素时，调用的是红黑树中的 insert_unique()方法，而非 insert_euqal()（multimap 使用）

举个例子:

```
Plaintext
#include <map>#include <iostream>#include <string>using namespace
std;int main(){        map<string, int> maps;    //插入若干元素
maps["jack"] = 1;        maps["jane"] = 2;        maps["july"] =
3;        //以 pair 形式插入        pair<string, int> p("david", 4);
maps.insert(p);        //迭代输出元素        map<string,
int>::iterator iter = maps.begin();        for (; iter !=
maps.end(); ++iter)        {                cout << iter->first <<
" ";                cout << iter->second << "--"; //david 4--jack
1--jane 2--july 3--        }        cout << endl;        //使用
subscipt 操作取实值        int num = maps["july"];        cout <<
```

```
num << endl; // 3        //查找某 key        iter =
maps.find("jane");        if(iter != maps.end())
cout << iter->second << endl; // 2    //修改实值        iter-
>second = 100;        int num2 = maps["jane"]; // 100        cout
<< num2 << endl;                return 0;}
```

需要注意的是 subscript（下标）操作既可以作为左值运用（修改内容）也可以作为右值运用（获取实值）。例如：

```
Plaintext
maps["abc"] = 1; //左值运用 int num = masp["abd"]; //右值运用
```

无论如何，subscript 操作符都会先根据键值找出实值，源码如下：

```
Plaintext
...T& operator[](const key_type& k){        return
(*((insert(value_type(k, T()))).first)).second;}...
```

代码运行过程是：首先根据键值和实值做出一个元素，这个元素的实值未知，因此产生一个与实值型别相同的临时对象替代：

```
Plaintext
value_type(k, T());
```

再将这个对象插入到 map 中，并返回一个 pair：

```
Plaintext
pair<iterator,bool> insert(value_type(k, T()));
```

pair 第一个元素是迭代器，指向当前插入的新元素，如果插入成功返回 true，此时对应左值运用，根据键值插入实值。插入失败（重复插入）返回 false，此时返回的是已经存在的元素，则可以取到它的实值

```
Plaintext
(insert(value_type(k, T()))).first; //迭代器*((insert(value_type(k,
T()))).first); //解引用(*((insert(value_type(k,
T()))).first)).second; //取出实值
```

由于这个实值是以引用方式传递，因此作为左值或者右值都可以

36、set 和 map 的区别，multimap 和 multiset 的区别

set 只提供一种数据类型的接口，但是会将这一个元素分配到 key 和 value 上，而且它

的 compare_function 用的是 identity()函数，这个函数是输入什么输出什么，这样就实现了 set 机制，set 的 key 和 value 其实是一样的了。其实他保存的是两份元素，而不是只保存一份元素

map 则提供两种数据类型的接口，分别放在 key 和 value 的位置上，他的比较 function 采用的是红黑树的 comparefunction（），保存的确实是两份元素。

他们两个的 insert 都是采用红黑树的 insert_unique() 独一无二的插入 。

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。

红黑树概念

面试时候现场写红黑树代码的概率几乎为 0，但是红黑树一些基本概念还是需要掌握的。

1、它是二叉排序树（继承二叉排序树特显）：

- 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、STL 中 unordered_map 和 map 的区别和应用场景

map 支持键值的自动排序，底层机制是红黑树，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

unordered_map 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

从两者的底层机制和特点可以看出：map 适用于有序数据的应用场景，
unordered_map 适用于高效查询的应用场景

38、hashtable 中解决冲突有哪些方法？

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

2.1.5、其余问题

1、C++的多态如何实现

C++的多态性，**一言以蔽之**就是：

在基类的函数前加上 **virtual** 关键字，在派生类中重写该函数，运行时将会根据所指对象的实际类型来调用相应的函数，如果对象类型是派生类，就调用派生类的函数，如果对象类型是基类，就调用基类的函数。

举个例子：

```
Plaintext
#include <iostream>using namespace std;class Base{public:
virtual void fun(){                cout << " Base::func()"
<<endl;        }};class Son1 : public Base{public:        virtual
void fun() override{                cout << " Son1::func()"
<<endl;        }};class Son2 : public Base{};int
main(){        Base* base = new Son1;        base->fun();
base = new Son2;        base->fun();        delete base;
base = NULL;        return 0;}// 运行结果// Son1::func()//
Base::func()
```

例子中，Base 为基类，其中的函数为虚函数。子类 1 继承并重写了基类的函数，子类 2 继承基类但没有重写基类的函数，从结果分析子类体现了多态性，那么为什么会出

现多态性，其底层的原理是什么？这里需要引出虚表和虚基表指针的概念。

虚表：虚函数表的缩写，类中含有 virtual 关键字修饰的方法时，编译器会自动生成虚表

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

上图中展示了虚表和虚表指针在基类对象和派生类对象中的模型，下面阐述实现多态的过程：

（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（派生类的虚函数表中存储的指针指向基类的虚函数的地址）；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

这样指向派生类的基类指针在运行时，就可以根据派生类对虚函数重写情况动态的进行调用，从而实现多态性。

2、为什么析构函数一般写成虚函数

由于类的多态性，基类指针可以指向派生类的对象，如果删除该基类的指针，就会调用该指针指向的派生类析构函数，而派生类的析构函数又自动调用基类的析构函数，这样整个派生类的对象完全被释放。

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。

所以将析构函数声明为虚函数是十分必要的。在实现多态时，当用基类操作派生类，在析构时防止只析构基类而不析构派生类的状况发生，要将基类的析构函数声明为虚函数。

```
Plaintext
#include <iostream>using namespace std;class Parent{public:
Parent(){                cout << "Parent construct function"  <<
endl;        };        ~Parent(){                cout << "Parent
destructor function" <<endl;        }};class Son : public
Parent{public:        Son(){                cout << "Son construct
function"  << endl;        };        ~Son(){                cout
<< "Son destructor function" <<endl;        }};int
main(){        Parent* p = new Son();        delete p;        p =
NULL;        return 0;}//运行结果：//Parent construct function//Son
construct function//Parent destructor function
```

将基类的析构函数声明为虚函数：

```
Plaintext
#include <iostream>using namespace std;class Parent{public:
Parent(){                cout << "Parent construct function"  <<
endl;        };        virtual ~Parent(){                cout <<
"Parent destructor function" <<endl;        }};class Son : public
Parent{public:        Son(){                cout << "Son construct
function"  << endl;        };        ~Son(){                cout
<< "Son destructor function" <<endl;        }};int
main(){        Parent* p = new Son();        delete p;        p =
NULL;        return 0;}//运行结果：//Parent construct function//Son
construct function//Son destructor function//Parent destructor
function
```

但存在一种特例，在 `CRTP` 模板中，不应该将析构函数声明为虚函数，理论上所有的父类函数都不应 该声明为虚函数，因为这种继承方式，不需要虚函数表。

update1:https://github.com/forthespada/InterviewGuide/issues/2 ,由 `stanleyguo0207` 提出 - 2021.03.22

3、构造函数能否声明为虚函数或者纯虚函数，析构函数呢？

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。

构造函数：

- 构造函数不能定义为虚函数。在构造函数中可以调用虚函数，不过此时调用的是正在构造的类中的虚函数，而不是子类的虚函数，因为此时子类尚未构造好。
- 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。（虚函数的调用需要通过虚表指针来实现，而该指针存放在对象的内存空间中，如果构造函数声明为虚函数，那么此时对象还未创建，还没有内存空间，就谈不上去指向虚函数表找相应的要调用的虚函数了）

update1:https://github.com/forthespada/InterviewGuide/issues/2 ,由 `stanleyguo0207` 提出 - 2021.03.22

4、基类的虚函数表存放在内存的什么区，虚表指针 vptr 的初始化时间

首先整理一下虚函数表的特征：

- 虚函数表是全局共享的元素，即全局仅有一个，在编译时就构造完成
- 虚函数表类似一个数组，类对象中存储 vptr 指针，指向虚函数表，即虚函数表不是函数，不是程序代码，不可能存储在代码段
- 虚函数表存储虚函数的地址,即虚函数表的元素是指向类成员函数的指针,而类中虚函数的个数在编译时期可以确定，即虚函数表的大小可以确定,即大小是在编译时期确定的，不必动态分配内存空间存储虚函数表，所以不在堆中

根据以上特征，虚函数表类似于类中静态成员变量.静态成员变量也是全局共享，大小确定，因此最有可能存在全局数据区，测试结果显示：

虚函数表 vtable 在 Linux/Unix 中存放在可执行文件的只读数据段中(rodata)，这与微软的编译器将虚函数表存放在常量段存在一些差别

由于虚表指针 vptr 跟虚函数密不可分，对于有虚函数或者继承于拥有虚函数的基类，对该类进行实例化时，在构造函数执行时会对虚表指针进行初始化，并且存在对象内存布局的最前面。

一般分为五个区域：栈区、堆区、函数区（存放函数体等二进制代码）、全局静态区、

常量区

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。**

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，举例如下：

```
Plaintext
template<typename T> //模板函数 int compare(const T &v1,const T &v2){    if(v1 > v2) return -1;    if(v2 > v1) return 1;    return 0;}//模板特例化,满足针对字符串特定的比较，要提供所有实参，这里只有一个 Ttemplate<> int compare(const char* const &v1,const char* const &v2){    return strcmp(p1,p2);}
```

本质

特例化的本质是实例化一个模板，而非重载它。特例化不影响参数匹配。参数匹配都以最佳匹配为原则。例如，此处如果是 compare(3,5)，则调用普通的模板，若为 compare(“hi”,”haha”)则调用**特例化版本**（因为这个 cosnt char*相对于 T，更匹配实参类型），注意二者函数体的语句不一样了，实现不同功能。

注意

模板及其特例化版本应该声明在同一个头文件中，且所有同名模板的声明应该放在前面，后面放特例化版本。

（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，例如：

```
Plaintext
template<>class hash<sales_data>{        size_t
```

```
operator()(sales_data& s);          //里面所有 T 都换成特例化类型版本
sales_data          //按照最佳匹配原则，若 T != sales_data，就用普通类模板，否则，就使用含有特定功能的特例化版本。};
```

类模板的部分特例化

不必为所有模板参数提供实参，可以**指定一部分而非所有模板参数**，一个类模板的部分特例化本身仍是一个模板，使用它时还必须为其特例化版本中未指定的模板参数提供实参(特例化时类名一定要和原来的模板相同，只是参数类型不同，按最佳匹配原则，哪个最匹配，就用相应的模板)

特例化类中的部分成员

可以特例化类中的部分成员函数而不是整个类，举个例子:

```
Plaintext
template<typename T>class Foo{    void Bar();    void Barst(T a)();};template<>void Foo<int>::Bar(){    //进行 int 类型的特例化处理 cout << "我是 int 型特例化" << endl;}Foo<string> fs;Foo<int> fi;//使用特例化 fs.Bar();//使用的是普通模板，即 Foo<string>::Bar()fi.Bar();//特例化版本，执行 Foo<int>::Bar()//Foo<string>::Bar()和 Foo<int>::Bar()功能不同
```

6、构造函数、析构函数、虚函数可否声明为内联函数

首先，将这些函数声明为内联函数，在语法上没有错误。因为 inline 同 register 一样，只是个建议，编译器并不一定真正的内联。

register 关键字：这个关键字请求编译器尽可能的将变量存在 CPU 内部寄存器中，而不是通过内存寻址访问，以提高效率

举个例子:

```
Plaintext
#include <iostream>using namespace std;class A{public:    inline A() {                cout << "inline construct()" <<endl;        } inline ~A() {                cout << "inline destruct()" <<endl;        }    inline virtual void  virtualFun()  {                cout << "inline virtual function" <<endl;        }}; int main(){        A a;        a.virtualFun(); return 0;}//输出结果//inline construct()//inline virtual function//inline destruct()
```

构造函数和析构函数声明为内联函数是没有意义的

《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**。其次，class 中的函数默认是 inline 型的，编译器也只是有选择性的 inline，将构造函数和析构函数声明为内联函数是没有什么意义的。

将虚函数声明为 inline，要分情况讨论

有的人认为虚函数被声明为 inline，但是编译器并没有对其内联，他们给出的理由是 inline 是编译期决定的，而虚函数是运行期决定的，即在不知道将要调用哪个函数的情况下，如何将函数内联呢？

上述观点看似正确，其实不然，如果虚函数在编译器就能够决定将要调用哪个函数时，就能够内联，那么什么情况下编译器可以确定要调用哪个函数呢，答案是当用对象调用虚函数（此时不具有多态性）时，就内联展开

综上，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，你知道底层怎么实现的？

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为虚函数？析构函数为什么要虚函数？

1、从存储空间角度，虚函数相应一个指向 vtable 虚函数表的指针，这大家都知道，但是这个指向 vtable 的指针事实上是存储在对象的内存空间的。

问题出来了，假设构造函数是虚的，就须要通过 vtable 来调用，但是对象还没有实例化，也就是内存空间还没有，怎么找 vtable 呢？所以构造函数不能是虚函数。

2、从使用角度，虚函数主要用于在信息不全的情况下，能使重载的函数得到相应的调用。

构造函数本身就是要初始化实例，那使用虚函数也没有实际意义呀。

所以构造函数没有必要是虚函数。虚函数的作用在于通过父类的指针或者引用来调用它的时候可以变成调用子类的那个成员函数。而构造函数是在创建对象时自己主动调

用的，不可能通过父类的指针或者引用去调用，因此也就规定构造函数不能是虚函数。

3、构造函数不须要是虚函数，也不同意是虚函数，由于创建一个对象时我们总是要明白指定对象的类型，虽然我们可能通过实验室的基类的指针或引用去訪问它但析构却不一定，我们往往通过基类的指针来销毁对象。这时候假设析构函数不是虚函数，就不能正确识别对象类型从而不能正确调用析构函数。

4、从实现上看，vbtl 在构造函数调用后才建立，因而构造函数不可能成为虚函数从实际含义上看，在调用构造函数时还不能确定对象的真实类型（由于子类会调父类的构造函数）；并且构造函数的作用是提供初始化，在对象生命期仅仅运行一次，不是对象的动态行为，也没有必要成为虚函数。

5、当一个构造函数被调用时，它做的首要的事情之中的一个是初始化它的 VPTR。

因此，它仅仅能知道它是“当前”类的，而全然忽视这个对象后面是否还有继承者。当编译器为这个构造函数产生代码时，它是为这个类的构造函数产生代码——既不是为基类，也不是为它的派生类（由于类不知道谁继承它）。所以它使用的 VPTR 必须是对于这个类的 VTABLE。

并且，仅仅要它是最后的构造函数调用，那么在这个对象的生命期内，VPTR 将保持被初始化为指向这个 VTABLE, 但假设接着另一个更晚派生的构造函数被调用，这个构造函数又将设置 VPTR 指向它的 VTABLE，等.直到最后的构造函数结束。

VPTR 的状态是由被最后调用的构造函数确定的。这就是为什么构造函数调用是从基类到更加派生类顺序的还有一个理由。可是，当这一系列构造函数调用正发生时，每一个构造函数都已经设置 VPTR 指向它自己的 VTABLE。假设函数调用使用虚机制，它将仅仅产生通过它自己的 VTABLE 的调用，而不是最后的 VTABLE（全部构造函数被调用后才会有最后的 VTABLE）。

因为构造函数本来就是为了明确初始化对象成员才产生的，然而 virtual function 主要是为了再不完全了解细节的情况下也能正确处理对象。另外，virtual 函数是在不同类型的对象产生不同的动作，现在对象还没有产生，如何使用 virtual 函数来完成你想完成的动作。

直接的讲，C++中基类采用 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。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。

8、析构函数的作用，如何起作用？

1. 构造函数只是起初始化值的作用，但实例化一个对象的时候，可以通过实例去传

递参数，从主函数传递到其他的函数里面，这样就使其他的函数里面有值了。

规则，只要你一实例化对象，系统自动回调用一个构造函数就是你不写，编译器也自动调用一次。

1. 析构函数与构造函数的作用相反，用于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；特点：析构函数与构造函数同名，但该函数前面加~。

析构函数没有参数，也没有返回值，而且不能重载，在一个类中只能有一个析构函数。当撤销对象时，编译器也会自动调用析构函数。

每一个类必须有一个析构函数，用户可以自定义析构函数，也可以是编译器自动生成默认的析构函数。一般析构函数定义为类的公有成员。

9、构造函数和析构函数可以调用虚函数吗，为什么

1. 在 C++中，提倡不在构造函数和析构函数中调用虚函数；

2. 构造函数和析构函数调用虚函数时都不使用动态联编，如果在构造函数或析构函数中调用虚函数，则运行的是为构造函数或析构函数自身类型定义的版本；

3. 因为父类对象会在子类之前进行构造，此时子类部分的数据成员还未初始化，因此调用子类的虚函数时不安全的，故而 C++不会进行动态联编；

4. 析构函数是用来销毁一个对象的，在销毁一个对象时，先调用子类的析构函数，然后再调用基类的析构函数。所以在调用基类的析构函数时，派生类对象的数据成员已经销毁，这个时候再调用子类的虚函数没有任何意义。

10、构造函数、析构函数的执行顺序？构造函数和拷贝构造的内部都干了啥？

1) 构造函数顺序

① 基类构造函数。如果有多个基类，则构造函数的调用顺序是某类在类派生表中出现的顺序，而不是它们在成员初始化表中的顺序。

② 成员类对象构造函数。如果有多个成员类对象则构造函数的调用顺序是对象在类中被声明的顺序，而不是它们出现在成员初始化表中的顺序。

③ 派生类构造函数。

2) 析构函数顺序

① 调用派生类的析构函数；

② 调用成员类对象的析构函数；

③ 调用基类的析构函数。

11、虚析构函数的作用，父类的析构函数是否要设置为虚函数?

1. C++中基类采用 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。

所以，为了防止这种情况的发生，C++中基类的析构函数应采用 virtual 虚析构函数。

1. 纯虚析构函数一定得定义，因为每一个派生类析构函数会被编译器加以扩张，以静态调用的方式调用其每一个虚基类以及上一层基类的析构函数。

因此，缺乏任何一个基类析构函数的定义，就会导致链接失败，最好不要把虚析构函数定义为纯虚析构函数。

12、构造函数析构函数可否抛出异常

1. C++只会析构已经完成的对象，对象只有在其构造函数执行完毕才算是完全构造妥当。在构造函数中发生异常，控制权转出构造函数之外。

因此，在对象 b 的构造函数中发生异常，对象 b 的析构函数不会被调用。因此会造成内存泄漏。

1. 用 auto_ptr 对象来取代指针类成员，便对构造函数做了强化，免除了抛出异常时发生资源泄漏的危机，不再需要在析构函数中手动释放资源;

2. 如果控制权基于异常的因素离开析构函数，而此时正有另一个异常处于作用状态，C++会调用 terminate 函数让程序结束;

3. 如果异常从析构函数抛出，而且没有在当地进行捕捉，那个析构函数便是执行不全的。如果析构函数执行不全，就是没有完成他应该执行的每一件事情。

13、构造函数一般不定义为虚函数的原因

（1）创建一个对象时需要确定对象的类型，而虚函数是在运行时动态确定其类型的。在构造一个对象时，由于对象还未创建成功，编译器无法知道对象的实际类型

（2）虚函数的调用需要虚函数表指针 vptr，而该指针存放在对象的内存空间中，若构造函数声明为虚函数，那么由于对象还未创建，还没有内存空间，更没有虚函数表 vtable 地址用来调用虚构造函数了

（3）虚函数的作用在于通过父类的指针或者引用调用它的时候能够变成调用子类的那个成员函数。而构造函数是在创建对象时自动调用的，不可能通过父类或者引用去调

用，因此就规定构造函数不能是虚函数

（4）析构函数一般都要声明为虚函数，这个应该是老生常谈了，这里不再赘述

14、类什么时候会析构?

1. 对象生命周期结束，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，而其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、构造函数或者析构函数中可以调用虚函数吗

简要结论：

- 从语法上讲，调用完全没有问题。
- 但是从效果上看，往往不能达到需要的目的。

《Effective C++》的解释是：派生类对象构造期间进入基类的构造函数时，对象类型变成了基类类型，而不是派生类类型。同样，进入基类析构函数时，对象也是基类类型。

举个例子：

```
Plaintext
#include<iostream>using namespace std;class Base{public:    Base()
{       Function();    }    virtual void Function()
{        cout << "Base::Fuction" << endl;    }        ~Base()
{               Function();        }};class A : public
Base{public:    A()    {      Function();    }    virtual void
Function()    {        cout << "A::Function" << endl;    }
~A()        {               Function();        }};int
main(){    Base* a = new Base;        delete a;        cout << "--
----------------------" <<endl;        Base* b = new A;//语句1
delete b;}//输出结果//Base::Fuction//Base::Fuction//---------------
----------//Base::Fuction//A::Function//Base::Fuction
```

语句 1 讲道理应该体现多态性，执行类 A 中的构造和析构函数，从实验结果来看，语句 1 并没有体现，执行流程是先构造基类，所以先调用基类的构造函数，构造完成再执行 A 自己的构造函数，析构时也是调用基类的析构函数，也就是说构造和析构中调用虚函数并不能达到目的，应该避免

16、构造函数的几种关键字

default

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

```
Plaintext
#include <iostream>using namespace std;class CString{public:
CString() = default; //语句 1    //构造函数    CString(const char*
pstr) : _str(pstr){}    void* operator new() = delete;//这样不允许
使用 new 关键字    //析构函数    ~CString(){}public:    string
_str;};int main(){   auto a = new CString(); //语句 2   cout <<
"Hello World" <<endl;   return 0;}//运行结果//Hello World
```

如果没有加语句 1，语句 2 会报错，表示找不到参数为空的构造函数，将其设置为 default 可以解决这个问题

delete

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

```
Plaintext
#include <iostream>using namespace std;class CString{public:
void* operator new() = delete;//这样不允许使用 new 关键字    //析构函
数    ~CString(){}};int main(){   auto a = new CString(); //语句 1
cout << "Hello World" <<endl;   return 0;}
```

在执行语句 1 时，会提示 new 方法已经被删除，如果将 new 设置为私有方法，则会报惨不忍睹的错误，因此使用 delete 关键字可以更加人性化的删除一些默认方法

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，不过函数体必须定义在类的外部）

17、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

举个例子：

```
Plaintext
#include <iostream>using namespace std;class A{public:        A()
{                cout << "我是构造函数" << endl;        }
A(const A& a)        {                cout << "我是拷贝构造函数" <<
endl;        }        A& operator = (A& a)
{                cout << "我是赋值操作符" << endl;
return *this;        }        ~A() {};};int main(){        A a1;
//调用构造函数        A a2 = a1; //调用拷贝构造函数        a2 = a1;
//调用赋值操作符        return 0;}//输出结果//我是构造函数//我是拷贝构
造函数//我是赋值操作符
```

18、拷贝构造函数和赋值运算符重载的区别?

- 拷贝构造函数是函数，赋值运算符是运算符重载。
- 拷贝构造函数会生成新的类对象，赋值运算符不能。
- 拷贝构造函数是直接构造一个新的类对象，所以在初始化对象前不需要检查源对象和新建对象是否相同；赋值运算符需要上述操作并提供两套不同的复制策略，另外赋值运算符中如果原来的对象有内存分配则需要先把内存释放掉。
- 形参传递是调用拷贝构造函数（调用的被赋值对象的拷贝构造函数），但并不是所有出现"="的地方都是使用赋值运算符，如下：

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

注：类中有指针变量时要重写析构函数、拷贝构造函数和赋值运算符。

19、什么是虚拟继承

由于 C++支持多继承，除了 public、protected 和 private 三种继承方式外，还支持虚拟（virtual）继承，举个例子：

```
Plaintext
#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}
```

上述代码所体现的关系是，B 和 C 虚拟继承 A，D 又公有继承 B 和 C，这种方式是一种**菱形继承或者钻石继承**，可以用如下图来表示

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 bptr，如上图所示。如果既存在 vptr 又存在 bptr，某些编译器会将其优化，合并为一个指针。

20、什么情况会自动生成默认构造函数?

1. 带有默认构造函数的类成员对象，如果一个类没有任何构造函数，但它含有一个成员对象，而后者有默认构造函数，那么编译器就为该类合成出一个默认构造函数。

不过这个合成操作只有在构造函数真正被需要的时候才会发生;

如果一个类 A 含有多个成员类对象的话，那么类 A 的每一个构造函数必须调用每一个成员对象的默认构造函数而且必须按照类对象在类 A 中的声明顺序进行;

1. 带有默认构造函数的基类，如果一个没有任务构造函数的派生类派生自一个带有默认构造函数基类，那么该派生类会合成一个构造函数调用上一层基类的默认构造函数;

2. 带有一个虚函数的类

3. 带有一个虚基类的类

4. 合成的默认构造函数中，只有基类子对象和成员类对象会被初始化。所有其他的非静态数据成员都不会被初始化。

21、抽象基类为什么不能创建对象?

抽象类是一种特殊的类，它是为了抽象和设计的目的为建立的，它处于继承层次结构

的较上层。

1、抽象类的定义： 称带有纯虚函数的类为抽象类。

2、抽象类的作用： 抽象类的主要作用是将有关的操作作为结果接口组织在一个继承层次结构中，由它来为派生类提供一个公共的根，派生类将具体实现在其基类中作为接口的操作。所以派生类实际上刻画了一组子类的操作接口的通用语义，这些语义也传给子类，子类可以具体实现这些语义，也可以再将这些语义传给自己的子类。

3、 抽象类只能作为基类来使用，其纯虚函数的实现由派生类给出。如果派生类中没有重新定义纯虚函数，而只是继承基类的纯虚函数，则这个派生类仍然还是一个抽象类。如果派生类中给出了基类纯虚函数的实现，则该派生类就不再是抽象类了，它是一个可以建立对象的具体的类。

抽象类是不能定义对象的。一个纯虚函数不需要（但是可以）被定义。

4、纯虚函数定义 纯虚函数是一种特殊的虚函数，它的一般格式如下：

```
Plaintext
class <类名>   {   virtual <类型><函数名>(<参数表>)=0;   …   };
```

在许多情况下，在基类中不能对虚函数给出有意义的实现，而把它声明为纯虚函数，它的实现留给该基类的派生类去做。这就是纯虚函数的作用。 纯虚函数可以让类先具有一个操作名称，而没有操作内容，让派生类在继承时再去具体地给出定义。凡是含有纯虚函数的类叫做抽象类。这种类不能声明对象，只是作为基类为派生类服务。除非在派生类中完全实现基类中所有的的纯虚函数，否则，派生类也变成了抽象类，不能实例化对象。

5、纯虚函数引入原因 1、为了方便使用多态特性，我们常常需要在基类中定义虚拟函数。 2、在很多情况下，基类本身生成对象是不合情理的。例如，动物作为一个基类可以派生出老虎、孔 雀等子类，但动物本身生成对象明显不合常理。 为了解决上述问题，引入了纯虚函数的概念，将函数定义为纯虚函数（方法：virtual ReturnType Function()= 0;）。若要使派生类为非抽象类，则编译器要求在派生类中，必须对纯虚函数予以重载以实现多态性。同时含有纯虚函数的类称为抽象类，它不能生成对象。这样就很好地解决了上述两个问题。 例如，绘画程序中，shape 作为一个基类可以派生出圆形、矩形、正方形、梯形等， 如果我要求面积总和的话，那么会可以使用一个 shape * 的数组，只要依次调用派生类的 area()函数了。如果不用接口就没法定义成数组，因为既可以是 circle ,也可以是 square ,而且以后还可能加上 rectangle，等等.

6、相似概念 1、多态性

指相同对象收到不同消息或不同对象收到相同消息时产生不同的实现动作。C++支持两种多态性：编译时多态性，运行时多态性。 a.编译时多态性：通过重载函数实现 b.运行时多态性：通过虚函数实现。 2、虚函数 虚函数是在基类中被声明为 virtual，并在派生类中重新定义的成员函数，可实现成员函数的动态重载。 3、抽象类 包含

纯虚函数的类称为抽象类。由于抽象类包含了没有定义的纯虚函数，所以不能定义抽象类的对象。

7、模板类和模板函数的区别是什么?

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

8、多继承的优缺点，作为一个开发者怎么看待多继承

1. C++允许为一个派生类指定多个基类，这样的继承结构被称做多重继承。
2. 多重继承的优点很明显，就是对象可以调用多个基类中的接口;
3. 如果派生类所继承的多个基类有相同的基类，而派生类对象需要调用这个祖先类的接口方法，就会容易出现二义性
4. 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，使得多重继承类 Programmer_Author 只拥有 Person 类的一份拷贝。

9、模板和实现可不可以不写在一个文件里面？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。《C++编程思想》第 15 章(第 300 页)说明了原因：模板定义很特殊。由 template<…>处理的任何东西都意味着编译器在当时不为它分配存储空间，

它一直处于等待状态直到被一个模板实例告知。在编译器和连接器的某一处，有一机制能去掉指定模板的多重定义。所以为了容易使用，几乎总是在头文件中放置全部的模板声明和定义。

10、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

11、为什么拷贝构造函数必须传引用不能传值?

1. 拷贝构造函数的作用就是用来复制对象的，在使用这个对象的实例来初始化这个对象的一个新的实例。

2. 参数传递过程到底发生了什么？ 将地址传递和值传递统一起来，归根结底还是传递的是"值"(地址也是值，只不过通过它可以找到另一个值)！ a 值传递: 对于内置数据类型的传递时，直接赋值拷贝给形参(注意形参是函数内局部变量)； 对于类类型的传递时，需要首先调用该类的拷贝构造函数来初始化形参(局部对象)；

如 void foo(class_type obj_local){}, 如果调用 foo(obj); 首先 class_type obj_local(obj) , 这样就定义了局部变量 obj_local 供函数内部使用

b 引用传递: 无论对内置类型还是类类型，传递引用或指针最终都是传递的地址值！而地址总是指针类型(属于简单类型), 显然参数传递时，按简单类型的赋值拷贝，而不会有拷贝构造函数的调用(对于类类型). 上述 1) 2)回答了为什么拷贝构造函数使用值传递会产生无限递归调用，内存溢出。

拷贝构造函数用来初始化一个非引用类类型对象，如果用传值的方式进行传参数，那么构造实参需要调用拷贝构造函数，而拷贝构造函数需要传递实参，所以会一直递归。

12、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。

13、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，增大类;

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

3. 不能再是内敛的函数，因为内敛函数在编译阶段进行替代，而虚函数表示等待，在运行阶段才能确定到低是采用哪种函数，虚函数不能是内敛函数。

14、说一说你了解到的移动构造函数?

1. 有时候我们会遇到这样一种情况，我们用对象 a 初始化对象 b 后对象 a 我们就不在使用了，但是对象 a 的空间还在呀（在析构之前），既然拷贝构造函数，实际上就是把 a 对象的内容复制一份到 b 中，那么为什么我们不能直接使用 a 的空间呢？这样就避免了新的空间的分配，大大降低了构造的成本。这就是移动构造函数设计的初衷;

2. 拷贝构造函数中，对于指针，我们一定要采用深层复制，而移动构造函数中，对于指针，我们采用浅层复制;

3. C++引入了移动构造函数，专门处理这种，用 a 初始化 b 后，就将 a 析构的情况;

4. 与拷贝类似，移动也使用一个对象的值设置另一个对象的值。但是，又与拷贝不同的是，移动实现的是对象值真实的转移（源对象到目的对象）：源对象将丢失其内容，其内容将被目的对象占有。移动操作的发生的时候，是当移动值的对象是未命名的对象的时候。这里未命名的对象就是那些临时变量，甚至都不会有名称。典型的未命名对象就是函数的返回值或者类型转换的对象。使用临时对象的值初始化另一个对象值，不会要求对对象的复制：因为临时对象不会有其它使用，因而，它的值可以被移动到目的对象。做到这些，就要使用移动构造函数和移动赋值：当使用一个临时变量对象进行构造初始化的时候，调用移动构造函数。类似的，使用未命名的变量的值赋给一个对象时，调用移动赋值操作;

5.

```
Plaintext
Example6 (Example6&& x) : ptr(x.ptr)   {    x.ptr = nullptr;  }
```

```
// move assignment  Example6& operator= (Example6&& x)
{   delete ptr;    ptr = x.ptr;   x.ptr=nullptr;    return *this;}
```

15、 什么时候合成构造函数？都说一说，你知道的都说一下

1. 如果一个类没有任何构造函数，但他含有一个成员对象，该成员对象含有默认构造函数，那么编译器就为该类合成一个默认构造函数，因为不合成一个默认构造函数那么该成员对象的构造函数不能调用；

2. 没有任何构造函数的类派生自一个带有默认构造函数的基类，那么需要为该派生类合成一个构造函数，只有这样基类的构造函数才能被调用；

3. 带有虚函数的类，虚函数的引入需要进入虚表，指向虚表的指针，该指针是在构造函数中初始化的，所以没有构造函数的话该指针无法被初始化；

4. 带有一个虚基类的类

还有一点需要注意的是：

1. 并不是任何没有构造函数的类都会合成一个构造函数

2. 编译器合成出来的构造函数并不会显示设定类内的每一个成员变量

16、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；

4. 如果一个类没有拷贝构造函数，但是含有一个类类型的成员变量，该类型含有拷贝构造函数，此时编译器会为该类合成一个拷贝构造函数；

5. 如果一个类没有拷贝构造函数，但是该类继承自含有拷贝构造函数的基类，此时编译器会为该类合成一个拷贝构造函数；

6. 如果一个类没有拷贝构造函数，但是该类声明或继承了虚函数，此时编译器会为该类合成一个拷贝构造函数；

7. 如果一个类没有拷贝构造函数，但是该类含有虚基类，此时编译器会为该类合成一个拷贝构造函数；

17、构造函数的执行顺序是什么？

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；

18、一个类中的全部构造函数的扩展过程是什么？

1. 记录在成员初始化列表中的数据成员初始化操作会被放在构造函数的函数体内，并与成员的声明顺序为顺序；

2. 如果一个成员并没有出现在成员初始化列表中，但它有一个默认构造函数，那么默认构造函数必须被调用；

3. 如果 class 有虚表，那么它必须被设定初值；

4. 所有上一层的基类构造函数必须被调用；

5. 所有虚基类的构造函数必须被调用。

19、哪些函数不能是虚函数？把你知道的都说一说

1. 构造函数，构造函数初始化对象，派生类必须知道基类函数干了什么，才能进行构造；当有虚函数时，每一个类有一个虚表，每一个对象有一个虚表指针，虚表指针在构造函数中初始化；

2. 内联函数，内联函数表示在编译阶段进行函数体的替换操作，而虚函数意味着在运行期间进行类型确定，所以内联函数不能是虚函数；

3. 静态函数，静态函数不属于对象属于类，静态成员函数没有 this 指针，因此静态函数设置为虚函数没有任何意义。

4. 友元函数，友元函数不属于类的成员函数，不能被继承。对于没有继承特性的函数没有虚函数的说法。

5. 普通函数，普通函数不属于类的成员函数，不具有继承特性，因此普通函数没有虚函数。

20、什么是纯虚函数，与虚函数的区别

虚函数和纯虚函数区别?

- 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。

纯虚函数首先是虚函数，其次它没有函数体，取而代之的是用“=0”。

既然是虚函数，它的函数指针会被存在虚函数表中，由于纯虚函数并没有具体的函数体，因此它在虚函数表中的值就为 0，而具有函数体的虚函数则是函数的具体地址。

一个类中如果有纯虚函数的话，称其为抽象类。抽象类不能用于实例化对象，否则会报错。抽象类一般用于定义一些公有的方法。子类继承抽象类也必须实现其中的纯虚函数才能实例化对象。

举个例子:

```
C++
#include <iostream>using namespace std;class Base{public:
virtual void fun1()        {                cout << "普通虚函数" <<
endl;        }        virtual void fun2() = 0;        virtual
~Base() {}};class Son : public Base{public:        virtual void
fun2()         {                cout << "子类实现的纯虚函数" <<
endl;        }};int main(){        Base* b = new Son;        b-
>fun1(); //普通虚函数        b->fun2(); //子类实现的纯虚函数
return 0;}
```