小🐻的求职笔记

C++

一、基础知识

1、在**main**执行之前和之后执行的代码可能是什么？

🐻回答：

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
- 初始化静态变量和全局变量，即.data段的内容
- 将未初始化部分的全局变量赋上初值，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传递给main函数

main函数执行之后

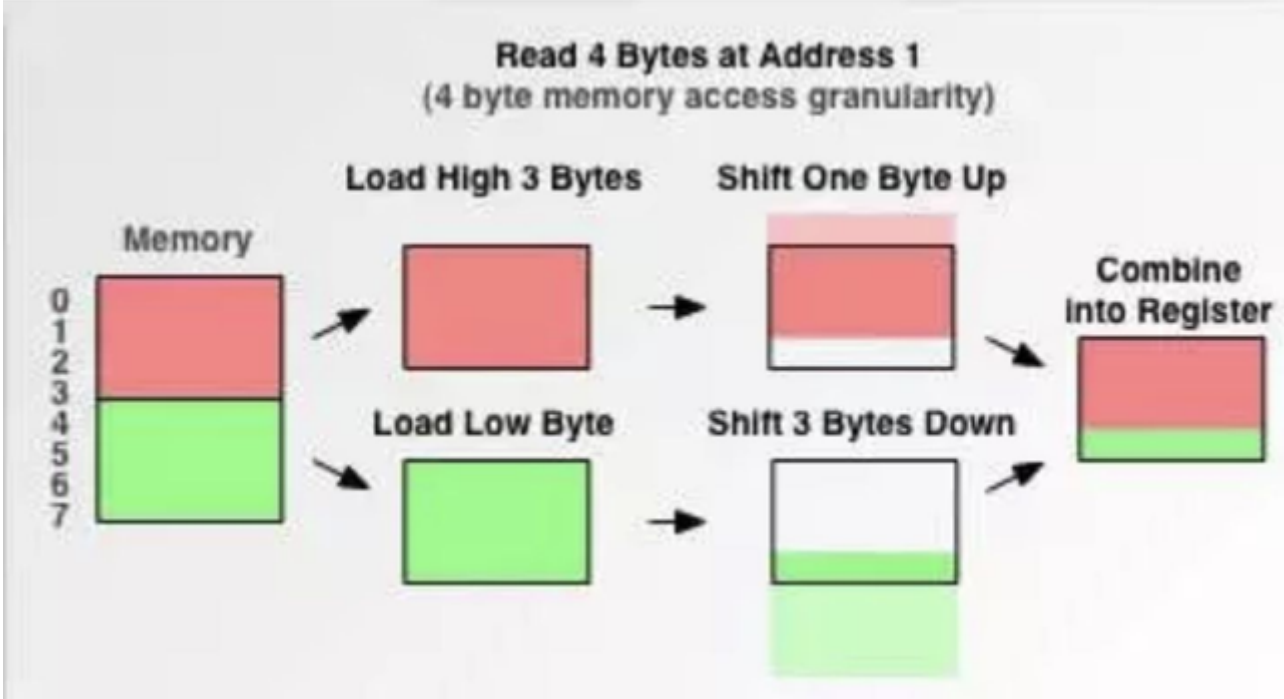
- 全局对象的析构函数会在main函数结束后执行

2、结构体内存对齐问题？

🐻回答：

为何需要内存对齐？

操作系统在数据读取的时候，其实并不是一个字节一个字节进行读取的，而是一段一段进行读取，我们假如是4bytes。假如我们要读取一个int，这个int是从第1位到第4位。那么读取的时候会发生什么事情呢？首先我们需要先读第一块数据，然后读取后三位的数据。接下来，读取第二块数据，然后只取第一位的数据。最后将两次的数据组合起来，就是我们想要的一个数据。



> *1*、第一个成员的首地址为*0*.
>
> *2*、每个成员的首地址是自身大小的整数倍
>
> *3*、结构体的总大小，为其成员中所含最大类型的整数倍。

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

3、指针和引用的区别？

回答：

- 指针是一个变量，存储的是一个地址。引用则是原变量的别名，和原变量实质上是一个东西。
- 指针可以有多级，而引用只有一级
- 指针可以为空，而引用必须在定义时初始化，不能为空
- sizeof指针得到的是指针的大小，而sizeof引用得到的是引用所绑定的变量的大小。
- 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了形参，两者指向相同，但并不是同一个指针。在函数中改变这个指针指向不影响实参。
- 指针声明和定义可以分开，但引用在声明时必须初始化。

- 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。

实际上，引用本身是一个变量，它底层是由一个常量指针的方式实现的。引用占用的内存和指针占用的内存长度一样，在 32 位环境下是 4 个字节，在 64 位环境下是 8 个字节，之所以不能获取引用的地址，是因为编译器进行了内部转换。

```
int a = 99;
int &r = a;
r = 18;
cout<<&r<<endl;
```

编译时会被转换成如下的形式：

```
int a = 99;
int *r = &a;
*r = 18;
cout<<r<<endl;
```

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？

回答：

- 函数不是构造函数，且参数是只读：用const引用。
- 参数是作为输出(out)参数：用指针。
- 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
- 作为类的拷贝构造函数的参数时，必须使用引用
- 当你有可能需要传入一个空的概念*（尚未创立的对象，空指针可以表示，而引用无法表示空的概念）的时候，指针才有必要。

5、堆和栈的区别

回答：

- 申请方式不同。
    - 栈由系统自动分配。
    - 堆是自己申请和释放的。
- 申请的大小限制不同

        - 栈顶和栈底都是之前预设好的，栈是向栈底扩展，大小固定（从高到低，低地址是栈顶）
        - 堆向高地址扩展，是不连续的内存区域，大小可以灵活调整。（从低到高）
    - 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。

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，向高地址方向增长。	栈向下，向低地址方向增长。

堆	栈	
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。

6、你觉得堆快一点还是栈快一点？

回答：

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？

回答：

```
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

回答：

相同点

- 都可用于内存的动态申请和释放

不同点

- 前者是C++关键字，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：

```
int *p = new float[2]; //编译错误
int *p = (int*)malloc(2 * sizeof(double));//编译无错误
```

- 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
- 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如何实现的？

回答：

- 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？

回答：

- 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（必须进行类型转换），new和delete返回的是具体类型指针。

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？

回答：

- 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回给os吗？

回答：

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、宏定义和函数的区别

回答：

- 宏在编译前（预编译）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的调用函数。
- 宏定义属于在结构体中插入代码，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

- 宏的参数是不占内存空间的,因为只是做字符串的替换,而函数调用时的参数传递则是具体变量之间的信息传递,形参作为函数的局部变量,显然是占用内存的.
- 宏定义不要在最后加分号

14、宏定义和**typedef**的区别

- 宏主要用于定义常量以及书写复杂的内容，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。
- `typedef char \* String_t` 定义了一个新的类型别名，有类型检查。
  `#define String_d char \*` 只做了简单的替换，无类型检查。
    - 前者在编译的时候处理，后者在预编译的时候处理。
    - 同时定义多个变量的时候有区别，主要区别在于这种使用方式：
      `String_t a,b;` `String_d c,d;`
      a，b，c 都是char *类型，而d为char类型

15、变量声明和定义的区别

回答：

- 变量可以声明多次，但只能定义一次
- 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。

16、**strlen**和**sizeof**的区别？

回答：

- sizeof是运算符而不是函数，结果在编译时得到而不是运行的时候，strlen是字符处理的库函数
- sizeof的参数可以是任何数据的类型或者数据，strlen参数只能是字符指针并且要以\0'结尾。
- 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）存储空间大小。
- sizeof算的是数据类型的大小，而strlen则是字符串的长度。

> 一个指针在*32*位环境占*4*字节，*64*位环境占*8*字节。

17、常量指针和指针常量的区别？

回答：

- 指针常量是一个指针，常量的指针，指向一个只读变量。也就是后面所指明的int const 和 const int，都是一个常量，可以写作int const *p或const int *p。
- 常量指针是一个不能改变指向的指针。指针是一个常量，必须初始化，一旦初始化完成，它的值（也就是存放在指针中的地址）就不能在改变了，即不能中途改变指向，如int *const p。

> 定义*const*变量后必须进行初始化。因为因为*const*变量不像普通的变量，不能够之后赋值。

18、a和&a的区别？

回答：

假设数组int a[10]; int (*p)[10] = &a;其中：

- 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与c++的区别

回答：

- 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++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）。

20、C++中struct和class的区别

回答：

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++的struct中能放成员函数，能继承，实现多态，C不行
- C中struct没有访问控制，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。
- 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。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。

21、define宏定义和const的区别

回答：

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

- 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。

22、C++中**const**和**static**的作用

回答：

详情见我的笔记——百面C++

23、C++顶层**const**和底层**const**

回答：

- 顶层const：指的是const修饰的变量本身是一个常量，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，指的是指针，就是 * 号的左边
- 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

24、数组名和指针（这里为指向数组首元素的指针）区别？

回答：

- 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，可以理解为常量指针，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。

25、**final**和**override**关键字

回答：

- 对虚函数重写时，后面加个override表示重写了
- 不希望某个虚函数被重写或者某个类被继承，就在后面加个final

26、拷贝初始化和直接初始化

回答：

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

- 直接初始化是根据参数寻找最优匹配构造函数的过程，所以上面两个都是直接初始化，但是第二个调用的是拷贝构造函数初始化。
- 拷贝初始化时，要求编译器将右侧运算对象拷贝到正在创建的对象中，如果需要还要进行类型转换。

  拷贝初始化不仅是=定义变量是会发生，在下列情况也会发生：

  - 将一个对象作为实参传递给一个非引用类型的形参
  - 从一个返回类型是非引用类型的函数返回一个对象
  - 用花括号列表初始化一个数组中的元素或一个聚合类的成员。

  拷贝初始化是使用拷贝构造函数实现的

  注意：

  - 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

27、初始化和赋值的区别

回答：

初始化是给一个刚定义的变量一个值。以确保程序的安全和确定。

赋值是给一个已经存在的变量一个值。
  ```

28、extern "C"的用法

回答：

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++

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。

C调用**C++**

```
/*C++ code*/
extern "C" void f(int);
void f(int i)
{
// your code
}

/*C code*/
void f(int);  // 不引入，而只是直接声明
void cc(int i)
{
f(i);  //调用
// other code
}
```

```
// C++ code:
class C
{
// ...
virtual double f(int);
};

extern "C" double call_C_f(C* p, int i) // wrapper function
{
return p->f(i);
}
/* C code: */
double call_C_f(struct C* p, int i);
```

```
void ccc(struct C* p, int i)
{
double d = call_C_f(p,i);
/* ... */
}
```

C++调用C

```
#include <iostream>
extern "C" {
    #include "add.h"  // 由add.h和add.c组成
}
using namespace std;

int main() {
    cout << addTwoNumber(10, 20) << endl;
    system("pause");
    return 0;
}
```

29、野指针和悬空指针

回答：

都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的行为。

- 野指针

  野指针，指的是没有被初始化过的指针

  ```
  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。

```
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。

30、C和C++的类型安全

回答：

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有授权的内存区域。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

```c
#include <stdio.h>

int main()
{
    printf("整型输出：%d\n",10);
    printf("浮点型输出：%f\n",10);
    return 0;
}
```

```
整型输出：10
浮点型输出：0.000000
```

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；他是进行const类型检查的。
- 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。

31、C++中的重载、重写和隐藏的区别

回答：

（1）重载

重载是指在同一范围定义中的同名成员函数才存在的重载关系。主要特点就是函数名相同。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。

（2）重写

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

（3）隐藏

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。
- 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。
- 补充

```
// 父类
class A {
public:
    virtual void fun(int a) { // 虚函数
        cout << "This is A fun " << a << endl;
    }
    void add(int a, int b) {
        cout << "This is A add " << a + b << endl;
    }
};

// 子类
class B: public A {
public:
    void fun(int a) override {  // 覆盖
        cout << "this is B fun " << a << endl;
    }
    void add(int a) {   // 隐藏
        cout << "This is B add " << a + a << endl;
    }
};

int main() {
    // 基类指针指向派生类对象时，基类指针可以直接调用到派生类的覆盖函数，也可以通过 :: 调用到基类被覆盖
```

```
    // 的虚函数；而基类指针只能调用基类的被隐藏函数，无法识别派生类中的
隐藏函数。

    A *p = new B();
    p->fun(1);      // 调用子类 fun 覆盖函数
    p->A::fun(1);   // 调用父类 fun
    p->add(1, 2);
    // p->add(1);       // 错误，识别的是 A 类中的 add 函数，参数不
匹配
    // p->B::add(1);    // 错误，无法识别子类 add 函数
    return 0;
}
```

32、C++中有哪几种构造函数？

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 （委托构造函数使用类的其他构造函数执行初始化过程）
- 转换构造函数

```
#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

#include<iostream>
using namespace std;
//类定义
class Clock {
public:
    //Clock() = default; //指示编译器提供默认构造函数
    Clock();//默认构造函数
    Clock(int newH, int newM, int newS);//构造函数
    void setTime(int newH, int newM, int newS);
    void showTime();
private:
    int hour, minute, second;
};
void Clock::setTime(int newH, int newM, int newS) {
    hour = newH;
    minute = newM;
```

```
    second = newS;
}
void Clock::showTime() {
    cout << hour << ":" << minute << ":" << second << endl;
}
//构造函数的实现:
Clock::Clock(int newH, int newM, int newS) :
    hour(newH), minute(newM), second(newS) {
}

//默认构造函数
//Clock::Clock() : hour(0), minute(0), second(0) { }

//委托构造函数
Clock::Clock():Clock(0,0,0){
}

int main() {
    Clock c1(11,11,11); //自动调用构造函数
    Clock c2;
    c1.showTime();
    c2.showTime();
    return 0;
}
```

33、深拷贝和浅拷贝的区别

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。

34、内联函数和宏定义的区别

- 内联函数具有类型检查功能，宏定义没有
- 宏 定义需要注意书写，不然会产生歧义

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。

35、public、protected、private

- 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。

36、说一下C++中的四种cast转换

static_cast

- 任何具有明确定义的类型转换，只要不包含底层const，都可使用

  > 底层*const*：指针所指对象为常量

- 例如，将较大的算术类型赋值给较小的类型，非const转为const，void*转指针。

const_cast

- 只能改变运算对象的底层const，将const转为非const。

dynamic_cast（简单介绍一下**RTTI**）

- 用于动态类型转换，只能用于含有虚函数的类，用于类层次间的向上或向下转换。只能转指针或引用。向下转换时，非法的对于指针返回NULL，对于引用抛出异常。
- 向上转换：子类转基类
- 向下转换：基类转子类。

RTTI

运行时类型识别（RTTI）的功能由两个运算符实现：

- typeid运算符，用于返回表达式的类型
- dynamic_cast运算符，用于将基类的指针或引用安全地转换成派生类的指针或引用。

我们将这两个运算符用于某种类型的指针或引用，如果这类型含有虚函数，则运算符将使用指针或引用所绑定对象的动态类型。

特别适用于以下情况：想使用基类对象的指针或引用执行某个派生类的操作并且该操作不是虚函数

一般来说我们尽可能使用虚函数，但没有只能使用一个RTTI运算符。

dynamic_cast运算符

举个栗子：

Base类至少含有一个虚函数，Derived是Base的公有派生类。如果有一个指向Base的指针bp，则我们可以在运行时将它转换成指向Derived的指针：

```
if (Derived *dp = dynamic_cast<Derived*>(bp)) {
    // 使用dp指向的Derived对象
} else {
    // 使用bp指向的Base对象
}
```

```
void f(const Base &b) {
    try {
        const Derived &d = dynamic_cast<const Derived&>(b);
    }catch(bad_cast) {
        // 处理类型转换失败的情况
    }
}
```

typeid运算符

typeid(e)，e为任意表达式或类型的名字。操作的结果是一个常量对象的引用，该对象的类型是标准库类型type_info或者是type_info的公有派生类型。

typeid运算符可以作用于任意类型的表达式。顶层const会被忽略。如果表达式是一个引用，typeid会返回该引用所引对象的类型。

如果是对象数组或函数操作，不会执行向指针的标准类型转换。对数组a执行typeid(a)，所得是数组类型而不是指针类型。

当运算对象不属于类类型或者没有任何虚函数的类，则指示的是运算对象的静态类型。当运算对象是定义了至少一个虚函数的类的左值时，**typeid**的结果到运行时才会求得

```
Derived *dp = new Derived;
Base *bp = dp;
if (typeid(*bp) == typeid(*dp)) {
    //
}
```

使用**RTTI**

```
class Base {
    friend bool operator==(const Base&, const Base&);
public:
    // Base的接口成员
protected:
    virtual bool equal(const Base&) const;
};
class Derived: public Base {
public:
    // Derived的接口成员
protected:
    bool equal(const Base&) const;
};

// 如果运算对象类型不同则返回false，否则当运算对象是Base，就调用Base的equal，是Derived就调用Derived的equal 比较成员。
bool operator==(const Base&, const Base&) {
    return typeid(lhs) == typeid(rhs) && lhs.equal(rhs);
}
bool Derived::equal(const Base& rhs) const
{
    // 我们清楚两个类型相等，所以转换不会抛出异常
    auto r = dynamic_cast<const Derived&>(rhs);
    // 执行比较操作并返回结果
}

bool Base::equal(const Base& rhs) {
    // 执行比较
```

```
}
```

reinterpret_cast

- 几乎什么类型都可以转换，比如int转指针，但可能会出问题，尽量不要使用。

37、C++中指针和引用的区别

- 指针有自己的一块空间，而引用只是一个别名
- sizeof指针是指针的大小，而sizeof引用是被引用对象的大小
- 指针可以被初始化为NULL，而引用必须被初始化且必须是一个已有对象的引用。
- 有const 指针，但没有const引用。
- 作为参数传递时，指针需要被解引用才可以对对象进行操作，而直接对引用的修改都会改变引用所指向的对象。
- 指针在使用中可以指向其他对象，而引用只能是一个对象的引用。不能被改变。
- 指针可以有多级指针（**p），而引用只有一级。

38、数组和指针，几种sizeof解析

- 数组不能拷贝。
- 指针保存数据的地址，数组保存数据。

首先数组名不是指针,这个可以用sizeof验证

`printf("%#x \n", &ca)`;是传递数组首地址,毫无疑问

`printf("%#x \n", ca)`;传递的是数组,但数组做参数传递时会自动退化成首地址,也就是说编译器自动加了个&

数组名被解释为其第一个元素的地址，而对数组名应用地址运算符（即&）时，得到的是整个数组的地址：

```
short tell[10];         //声明一个长度为20字节的数组（short型变量大小为2字节）
cout << tell << endl;   //显示&tell[0]
cout << &tell << endl; //显示整个数组的地址
```

从数字上说，这两个地址相同；但从概念上说，&tell[0]（即tell）是一个2字节内存块的地址，而&tell是一个20字节内存块的地址。因此，表达式tell+1将地址值加2，而表达式&tell+1将地址值加20。换句话说，tell是一个short指针（`short*`），而&tell是一个指向包含**10**个元素的**short**数组的指针（short(*) [10]）。

您可能会问，前面有关&tell的类型描述是如何来的呢？
首先，您可以这样声明和初始化这种指针：

```
short (*pas) [10] = &tell; //pas指向一个有10个short元素的数组
```

如果省略括号，优先级规则将使pas先与[10]结合，导致pas是一个包含**10**个**short**型指针的数组，因此括号是必不可少的。
其次，如果要描述变量的类型，可将声明中的变量名删除。因此，pas的类型为`short(*) [10]`。
另外，由于pas被设置为tell，因此`*pas`与tell等价，所以`(*pas) [0]`为tell数组的第一个元素。

39、为什么析构函数必须是虚函数？为什么**C++**默认的析构函数不是虚函数

- 将可能被继承的父类析构函数设为虚函数，当我们new一个子类，使用基类指针指向该子类对象，释放基类指针时能够释放子类的空间，防止内存泄漏。
- 默认的析构不是虚函数是因为需要额外的虚函数表和虚表指针，占用额外的内存，所以对于不会被继承的类来说，其析构函数如果是虚函数，就会浪费内存。因此C++默认的析构函数不是虚函数。而只有当需要当作父类时，才设置为虚函数。

40、**C++**析构函数的作用

- 当对象结束其生命周期，系统自动执行析构函数。
- 如果一个类有指针，且在使用中动态申请了内存，那么最好显示构造析构函数在销毁类之前，释放申请的内存空间，避免内存泄漏。

41、静态函数和虚函数的区别

- 静态函数在编译的时候就已经确定运行时机，虚函数在运行的时候动态绑定。

42、理解一下虚函数和多态

- 多态实现主要分为静态多态和动态多态，静态多态主要是重载，在编译的时候就已经确定，动态多态是用虚函数机制实现，在运行期间动态绑定。

43、C++中拷贝赋值函数的形参能否进行值传递？

- 不能，因为会递归调用，值传递的过程中又会调用拷贝赋值。

44、

```
const char* arr = "123";  // 字符串"123"在常量区
char* brr = "123"; // 用C++编译器通过不了
const char crr[] = "123";  // 不能修改，编译器可能会优化将其放到常量区
char drr[] = "123" // 字符串"123"保存在栈区，可以通过drr修改
```

C++中，string转换成C风格字符串都是const char*

45、C++一般如何定义常量？常量一般存放在哪个位置？

两种定义方法：

- 使用 **#define** 预处理器。
- 使用 **const** 关键字。

常量定义必须初始化。

const 如果修饰在函数参数或局部栈变量的话，那肯定存放在栈上无疑了。

对于全局对象，常量存放在全局/静态存储区，其实还是相当于变量，只不过这个变量readonly。

对于字面值常量，常量存放在常量存储区。

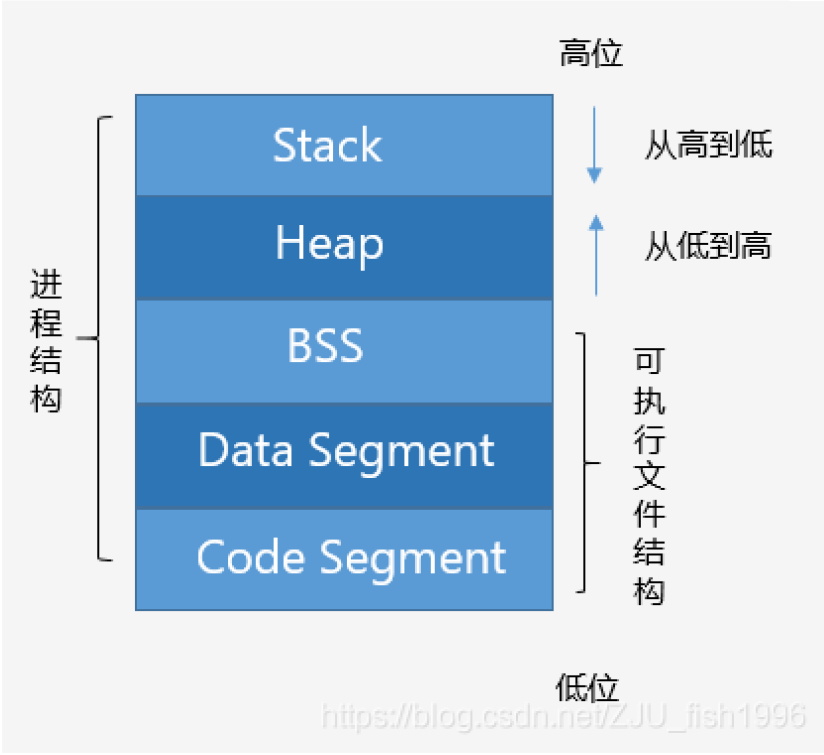
> 内存布局：
>
> - 代码区（*.text*）

- 全局/静态存储区*(.bss .rwdata)*
- 常量存储区*(.rodata)*
- 栈
- 堆

对应：

- 栈
- 堆
- *BSS(.bss)*
- *Data Segment(.rodata .rwdata)*
- *text*

内存布局



Code Segment（代码区）

也称Text Segment，存放可执行程序的机器码。

Data Segment（数据区）

存放已初始化的全局和静态变量，常量数据（如字符串常量）。

BSS（Block started by symbol）

存放未初始化的全局和静态变量。（默认设为0）

Heap（堆）

从低地址向高地址增长。容量大于栈，程序中动态分配的内存在此区域。

Stack（栈）

从高地址向低地址增长。由编译器自动管理分配。程序中的局部变量、函数参数值、返回变量等存在此区域。

函数栈

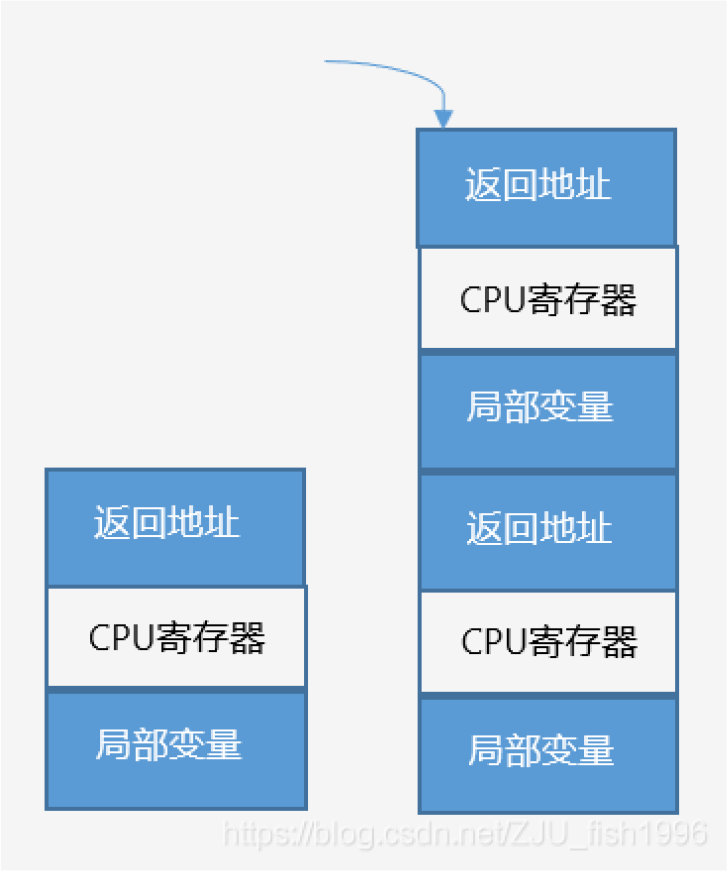
如上图所示，可执行程序的文件包含BSS，Data Segement和Code Segment，当可执行程序载入内存后，系统会保留一些空间，即堆区和栈区。堆区主要是动态分配的内存（默认情况下），而栈区主要是函数以及局部变量等（包括main函数）。一般而言，栈的空间小于堆的空间。

当调用函数时，一块连续内存(堆栈帧）压入栈；函数返回时，堆栈帧弹出。

堆栈帧包含如下数据:

① 函数返回地址

② 局部变量/CPU寄存器数据备份



46、如何用代码判断大小端存储？

回答：

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：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x78	0x56	0x34	0x12

大端模式中的存储方式为：

内存地址	0x4000	0x4001	0x4002	0x4003
存放内容	0x12	0x34	0x56	0x78

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

```
#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

```
#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 == 0x12)
        cout << "big endian"<<endl;
    else if (value.ch == 0x34)
        cout << "little endian"<<endl;
}
```

47、volatil、mutable和explicit关键字的用法

回答：

1、volatile

直接处理硬件的程序常常包含这样的数据元素，它们的值由程序直接控制之外的过程控制，例如，程序可能包含一个由系统时钟定时更新的变量。关键字volatile告诉编译器不应该对这样的对象进行优化。

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：

```
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：

```
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。

  > 因为合成的拷贝/赋值是无效的，如果想要能赋值，必须自定义。

- 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。

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”，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。

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，需要多个实参的构造函数不能用于隐式转换，所以无须指定。
- **explicit**只能在类内出现，在类外定义时不需要出现。

48、C++几种类型的**new**

回答：

（1）new

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

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

（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：

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

```
#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：

```
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。

```
#include <iostream>
#include <string>
using namespace std;
class ADT{
```

```
    int i;
    int j;
public:
    ADT(){
        i = 10;
        j = 100;
        cout << "ADT construct i=" << i << "j="<<j <<endl;
    }
    ~ADT(){
        cout << "ADT destruct" << endl;
    }
};
int main()
{
    char *p = new(nothrow) char[sizeof ADT + 1];
    if (p == NULL) {
        cout << "alloc failed" << endl;
    }
    ADT *q = new(p) ADT;  //placement new:不必担心失败，只要p所指对象的的空间足够ADT创建即可
    //delete q;//错误!不能在此处调用delete q;
    q->ADT::~ADT();//显示调用析构函数
    delete[] p;
    return 0;
}
//输出结果:
//ADT construct i=10j=100
//ADT destruct
```

48、C++的异常处理方法

回答：

（1）try、throw和catch关键字

```
#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
```

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

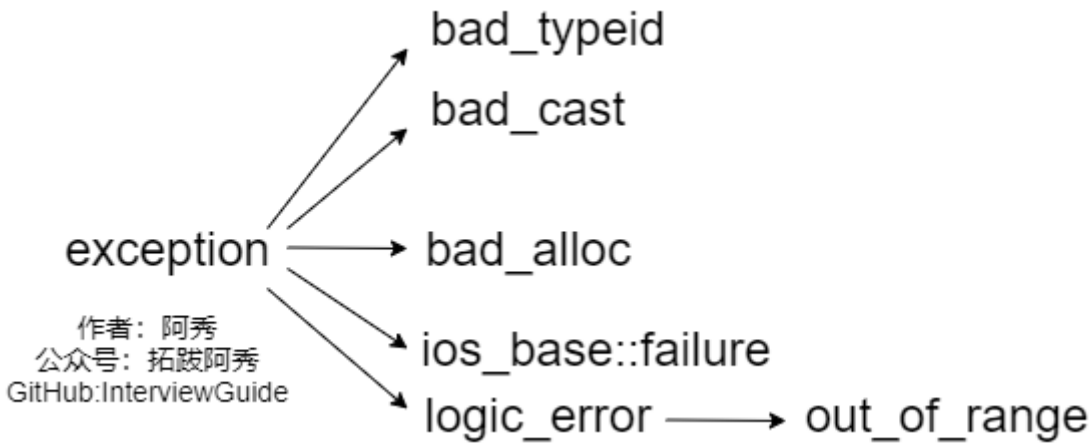
有时候，程序员在定义函数的时候知道函数可能发生的异常，可以在函数声明和定义时，指出所能抛出异常的列表，写法如下：

```
int fun() throw(int,double,A,B,C){...};Copy to clipboardErrorCopied
```

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：

```
#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：Object 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

49、形参与实参的区别？

回答：

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。

50、值传递、指针传递、引用传递的区别和效率

回答：

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。

51、静态变量什么时候初始化

回答：

静态变量存储在虚拟地址空间的数据段和bss段，c语言中其在代码执行之前初始化，属于编译期初始化，而c++中由于引入了对象，对象生成必须调用构造函数，因此c++中规定全局或者局部静态对象当且仅当对象首次用到时进行构造。

1. 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。

2. 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。

52、new和malloc的区别？

回答：

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。

```
char* a = (char*)malloc(20); // 20个字节
int* b = new int(15); // *b为15
```

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

回答：

1、 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；

2、 delete[]时，数组中的元素按逆序的顺序进行销毁；

3、 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如何知道释放内存的大小的？

回答：

1、new

new简单类型直接调用operator new分配内存。

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。

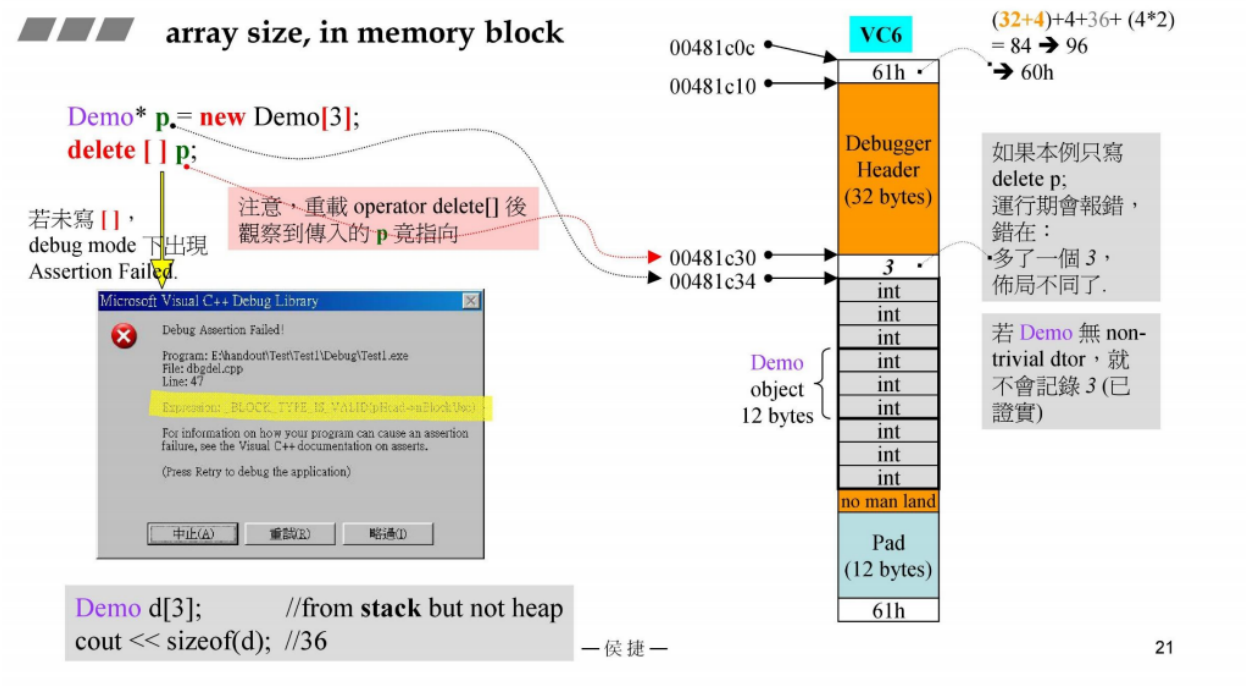
2、delete

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。



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申请的存储空间可以用delete释放吗？

回答：

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实现简单原理（暂且就这样）

回答：

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和calloc的区别

回答：

1. malloc函数

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

2. calloc函数

```
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；

3. realloc函数

```
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、类成员初始化的方式？构造函数的执行顺序？为什么用成员初始化列表快一些？

回答：

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。还有一种声明时初始化。

   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。

2. 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。

3. 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？

回答：

- 当初始化一个引用成员时

- 当初始化一个常量成员时
- 当调用一个基类的构造函数，而它拥有一组参数时
- 当调用一个成员类的构造函数，而它拥有一组参数时

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

回答：

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、什么是内存泄漏，如何避免？

回答：

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

- 智能指针管理
- 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

62、介绍面向对象三大特性，并且举例说明

回答：

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

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）。

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。

63、函数调用过程

回答：

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。

64、说说移动构造函数

回答：

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；
3. 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。

65、静态类型和动态类型，静态绑定和动态绑定

回答：

- 静态类型：对象在声明时采用的类型，在编译期已经确定。
- 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

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；
- 要想实现多态，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；

66、绝不重新定义继承而来的缺省参数值（待补充原因）

回答：

- 讨论范围：继承一个带有缺省参数值的virtual函数
- 注意：virtual函数是动态绑定，但缺省参数值却是静态绑定

```
class Shape {
public:
    enum ShapeColor { Red, Green, Blue};
    virtual void draw(ShapeColor color = Red) const = 0;
};
class Rectangle: public Shape {
public:
    virtual void draw(ShapeColor color = Green) const;
};
class Circle: public Shape {
public:
    virtual void draw(ShapeColor color)const;
    // 以上这么些则当以客户为对象调用此函数，必须要指定参数值
    // 静态绑定下这个函数并不从base继承缺省参数值
    // 动态绑定下这个函数会从base继承缺省参数值
};
Shape* ps;
Shape* pc = new Circle;
Shape* pr = new Rectangle;
pc->draw(Shape::Red); // 调用Circle::draw(Shape::Red)
pr->draw(Shape::Red); // 调用Rectangle::draw(Shape::Red)
// virtual函数是动态绑定，但缺省值则是静态绑定 调用一个定义于继承类的虚函数的同时，使用基类为它所指定的缺省参数值
pr->draw();   // 调用Rectangle::draw(Shape::Red)
```

```
// Rectangle的缺省值应该是Green，但由于pr的静态类型是Shape*，所以此一调用的缺省参数值来自Shape类。
```

缺省参数值总是由静态类型提供！

67、全局变量和局部变量有什么区别？

回答：

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

68、C++中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？

回答：

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。

值传递的过程中，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主调函数传递过来的实参值，从而形成了实参的一个副本。

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。

被调函数对形参的任何操作都被处理为间接寻址，即通过栈中存放的地址访问主调函数中的实参变量。

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。

69、类如何实现只能静态分配和只能动态分配

回答：

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

70、如果想将某个类用作基类，为什么该类必须定义而非声明？

回答：

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。

71、函数指针？

1）什么是函数指针？

函数指针是一个指针，指向的是函数的类型，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。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，他指向函数的代码

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

72、函数调用过程栈的变化

回答：

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

4、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); esp -> ebp

> *esp*指向栈顶，*ebp*指向栈底

5、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;

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

回答：

一、const与#define的区别

- 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
- 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；
- define预处理后，占用代码段空间，const占用数据段空间

二、#define与typedef的区别

- 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；

- 功能差异，typedef用来定义类型的别名；#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。

三、define与inline的区别

- #define是关键字，inline是函数
- 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
- inline函数有类型检查，相比宏定义比较安全

74、为什么模板类一般都是放在一个h文件中

回答：

- 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。

75、你知道重载运算符吗？

回答：

- 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
- 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
- ```
  (1)"."（类成员访问运算符）
  (2)" .*"（类成员指针访问运算符）
  (3) "::"（域运算符）
  (4)"siezof"（长度运算符）
  (5) " ?:"（条件运算符）
  ```

  以上五个不能重载
  ```

76、静态成员与普通成员的区别是什么？

回答：

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；

2. 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；

3. 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；

4. 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；

5. 默认实参

   可以使用静态成员变量作为默认实参

77、说一下你理解的**ifdef endif**代表着什么？

回答：

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。

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。

78、隐式转换

回答：

- 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。

79、你知道**strcpy**和**memcpy**的区别是什么？

回答：

- 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
- 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。

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

回答：

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。

81、如果有一个空类，它会默认添加哪些函数？

回答：

```
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//
```

82、C++标准库是什么？

回答：

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。

> 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

面向对象类库： 这个库是类及其相关函数的集合。

> 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

83、成员初始化列表会在什么时候用到？

回答：

- 当初始化一个引用成员变量时
- 初始化一个const成员变量时
- 当调用一个基类的构造函数，而构造函数拥有一组参数时
- 当调用一个成员类的构造函数，而他拥有一组参数

84、你知道数组和指针的区别吗？

回答：

- 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)
- 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。
- 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
- 指针可以改变，但数组名相当于常量指针

85、如何组织一个类被实例化？

- 将类定义为抽象基类或者将构造函数声明为private

86、如何禁止程序自动生成拷贝构造函数？（修改）

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。

87、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。

88、static_cast比C语言中的转换强在哪里？

- 有类型检查，更加安全

89、你知道回调函数吗？它的作用是什么？

- 当有信号或者有事件发生的时候，系统或其他函数将会自动调用你定义的一段函数
- 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；
- 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；

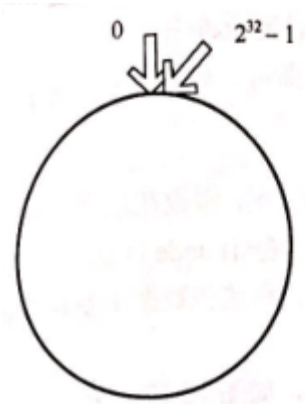
90、什么是一致性哈希？

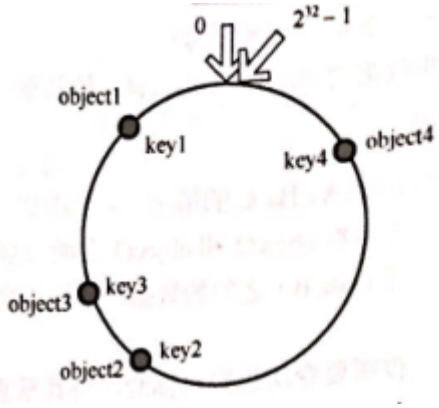
一致性哈希

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



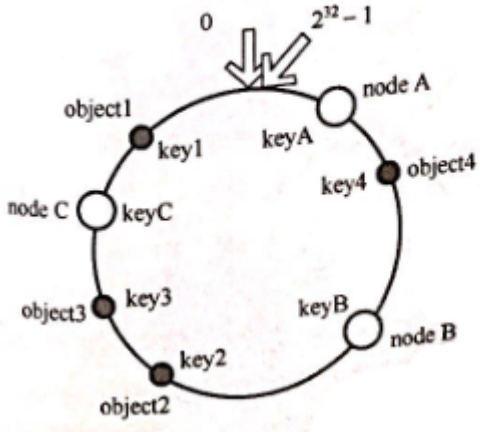


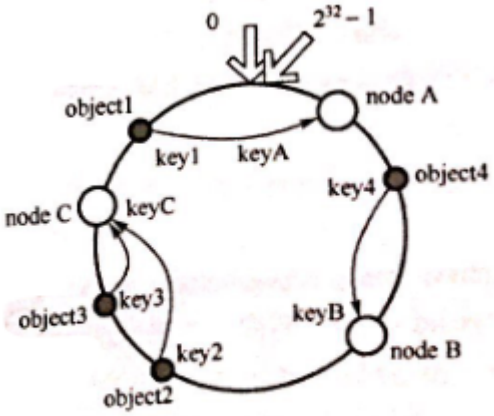
一致性hash的基本思想就是使用相同的hash算法将数据和结点都映射到图中的环形哈希空间中，上右图显示了4个数据object1-object4在环上的分布图

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

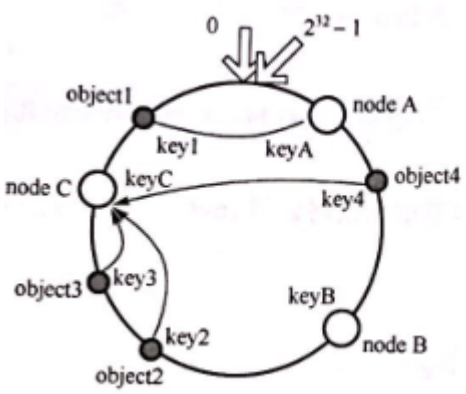
现在有一批的数据object1-object4需要存在服务器上，则可以使用相同的哈希算法对数据进行哈希，其结果必然也在环上，可以沿着顺时针方向寻找，找到一个结点（服务器）则将数据存在这个结点上，这样数据和结点就产生了一对一的关联，如下图所示：

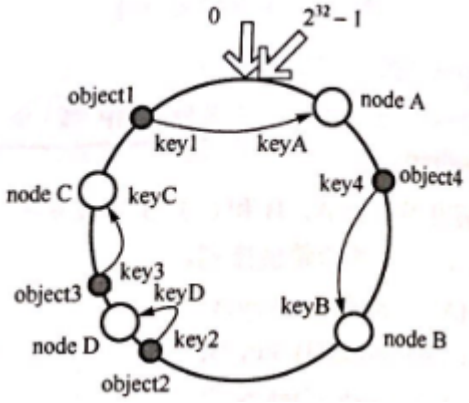




移除结点

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



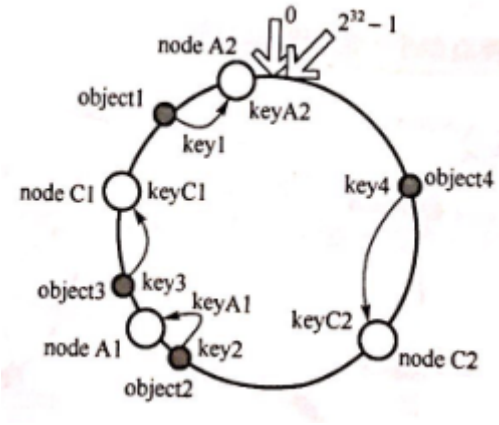


添加结点

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

91、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。处理规则如下：

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。

> *.i*文件

（**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。

> *.s*文件

（**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

> *.o*文件

（**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。

92、C语言模拟C++的继承与多态

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

typedef void (*FUN)();    //定义一个函数指针来实现对成员函数的继承
struct base
{
    FUN _fun;
    int a;
};

struct derived
{
    struct base a;
    int b;
};

void basefun()
{
    cout << "base" << endl;
}

void derivedfun()
{
    cout << "derivedfun" << endl;
}
int main() {
    base ba;
    derived da;
```

```
    ba._fun = basefun;
    da.a._fun = derivedfun;

    base* p = &ba;
    p->_fun();
    p = (base*)&da;
    p->_fun();

    return 0;
}
/*
base
derivedfun
*/
```

93、为什么不能把所有的函数写成内联？

- 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

二、C++11新标准

1、C++11有哪些新特性？

回答：

- nullptr替代NULL
- 引入了auto和decltype两个关键字实现了类型推导
- 基于范围的for循环
- 类和结构体中的列表初始化
- Lambda表达式（匿名函数）
- 右值引用和move语义

2、auto、decltype和decltype(auto)的用法？

回答：

（1）auto

C++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。

auto让编译器通过初始值来进行类型推导。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。

在**auto**的推断过程中，一般遵循四个原则：

- 同一条声明语句只能有一个基本数据类型，所以该语句中所有变量的初始基本数据类型都必须一致
- 符号&和*只从属于某个声明符，而非基本类型的一部分
- 当引用被当做初始值的时候，真正用于推断 **auto** 类型的初始值实际上是引用对象的值。
- 当用于推断 **auto** 类型的初始值是常量时，如果可以忽略其常量性质，则忽略其常量性质。

```
int i = 0;
const int c1 = i;
const int &c2 = i;
const int *p2 = &i;
int *const p3 = &i;

// 前两条规则的示例
auto j = 0, *p = &j;         //j的类型是int，p的类型是int *，该语句的基本数据类型是int
auto sz = 0, pi= 3.14;       //错误，sz和pi的类型不同
auto &n = c1, *p1 = &c1;     //n的类型是const int &，p1的类型是const int *，该语句的基本数据类型是const int

// 后两条规则的示例
auto a = i;      //i是一个int型变量，所以a是int型
auto b = c1;     //c1是一个const int型的常量，初始值本身是个常量，故忽略其常量性质，所以b是int型
auto c = c2;     //c2是一个const int &型的引用，所以实际上的初始值是引用对象c1，故c是int型
auto d = &c1;    //c1是一个const int型的常量，对其取地址得到的const int*型的初始值，注意该初始值本身并不是一个常量，而是其指向的对象是常量，故d是const int*型
```

```
auto e = p2;    //p2是一个const int*型，初始值并不是常量，故e是const int*型
auto f = p3;    //p3是一个int *类型的常量指针，所以忽略其常量性质，则f是int *类型
auto &g = c1;   //c1是一个const int型的常量，如果忽略其常量性质，则得到g是与从绑定的int &型变量，显然违背了基本原则，所以该处不能忽略c1的常量性质，则g是const int &类型
```

> 顶层*const* 是自身为常量。

（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。

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。

```
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一样，都是顶层const  const int
int e = 4;
const int* f = &e; // f也是底层const
decltype(f) g = f; // g为底层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; // 加法的结果是int，因此t是int类型，可以不初始化。

//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
```

auto会忽略顶层**const**和引用，但**decltype**不好会

（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：

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

3、C++中NULL和nullptr的区别？

回答：

算是为了与C语言兼容而定义的一个问题

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：

```
#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：

```
#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。

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：

```
#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

回答：

- 动态内存管理通过一对运算符来完成：new，在动态内存中为对象分配空间并返回一个指向该对象的指针。delete，接受一个动态对象的指针，销毁该对象，并释放与之关联的内存。
- 忘记释放内存，就会导致内存泄漏。
- 为了更容易且更安全使用动态内存，新的标准库提供了两种智能指针类型来管理动态对象。智能指针行为类似常规指针，重要的区别是它负责自动释放所指向的对象。因为智能指针就是一个类，当超出类的作用域，类就会自动调用析构函数释放资源。
- shared_ptr允许多个指针指向同一个对象，unique_ptr“独占”所指向的对象。
- weak_ptr（伴随类），指向shared_ptr所管理的对象。

auto_ptr

- auto_ptr被销毁时会自动删除它所指之物，所以一定要注意别让多个auto_ptr同时指向一个对象！
- 若选择auto_ptr，复制动作会使它指向null，而复制所得的指针将取得资源的唯一拥有权！

shared_ptr（实现）

- auto_ptr太菜了！还是shared_ptr比较好，实现共享式拥有概念，多个智能指针可以指向相同对象，该对象和其相关资源会在“最后一个引用被销毁”时候释放。解决了auto_ptr在对象所有权上的局限性（auto_ptr是独占的），在使用引用计数的机制上提供了可以共享所有权的智能指针。

```
shared_ptr<string>p1; // 指向string  默认初始化的智能指针保存着一个空指针。
```

  - 默认初始化的智能指针中保存着一个空指针
  - 解引用一个智能指针返回它指向的对象。

```
shared_ptr<T>p;     // 空智能指针，可以指向类型为T的对象
unique_ptr<T>up;
p;        //  将p作为一个条件判断，如果p指向一个对象，为true
*p;       //  解引用p，得到指向对象
p->mem;   //  (*p).mem
p.get(); // 返回p保存的指针，若智能指针释放了对象，返回的指针所指向的对象也就没了
swap(p,q); // 交换p和q中的指针
p.swap(q);// shared_ptr独有操作
p.unique() // 若p.use_count()为1 返回true
p.use_count() // 返回与p共享对象的智能指针数量
shared_ptr<string>p;
if (p && p->empty()) {
    *p = "hi";                  // p不为空且p指向的对象为空string，将“hi”赋给该对象
}
```

make_shared函数

- 最安全的分配和使用动态内存的方法是调用一个名为make_shared的标准库函数。此函数在动态内存中分配一个对象并初始化它，返回此对象的shared_ptr。
- make_shared(args) 用args来初始化对象
- ```
  shared_ptr<int>p3 = make_shared<int>(42);  // 指向一个值为42的int的shared_ptr
  shared_ptr<string>p4 = make_shared<string>(10,'9');
   // 指向一个值为"9999999999"的string的shared_ptr
  shared_ptr<int>p5 = make_shared<int>();  // 指向一个值初始化为0的int的shared_ptr
  ```
  ```

- make_shared用其参数来构造给定类型的对象。

```
shared_ptr<T>p(q,d); // p接管了内置指针q所指向的对象的所有权。q必须能转换为T*类型，p将使用可调用对象d来代替delete
p.reset() // 若p是唯一指向其对象的shared_ptr，reset会释放此对象。若传递了内置指针q，将p中内置指针换为q，否则会让p置为空
p.reset(q);
p.reset(q,d); // 若还有参数d，会调用d而不是delete。
```

- 实现，见我的项目

weak_ptr

- 是一种不控制所指向对象生存期的智能指针。它指向一个shared_ptr管理的对象。将一个weak_prt绑定到一个shared_ptr不会改变shared_ptr的引用计数。
- 一旦最后一个指向对象的shared_ptr被销毁，对象就会被释放，即使有weak_ptr指向对象，对象还是被释放。

```
weak_ptr<T>w(sp); // 与shared_ptr sp指向相同对象的weak_ptr。T必须能转换为sp指向的类型。
w = p; // p可以说shared_ptr 或 weak_ptr，赋值后w和p共享对象
w.reset(); // 将w置空
w.use_count(); // 与w共享对象的shared_ptr数量
w.expired(); // w.use_count() 为0 返回true
w.lock(); // 如果expired为true，返回一个空shared_ptr 否则返回一个指向共享对象的shared_ptr
```

创建一个**weak_ptr**，需要用**shared_ptr**来初始化。

- weak_ptr只是提供了对管理对象的一个访问手段。**weak_ptr**设计的目的是为了配合**shared_ptr**，它只可以从一个shared_ptr或者weak_ptr对象来构造。它的构造和析构不会引起引用计数的增加或减少。
- 可以解决循环引用问题

  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：

```
#include <bits/stdc++.h>
using namespace std;
struct ListNode
{
    int _data;
```

```
    shared_ptr<ListNode> _prev;
    shared_ptr<ListNode> _next;
    ~ListNode() {
        cout << "~ListNode()" << endl;
    }
};

int main()
{
    shared_ptr<ListNode> node1(new ListNode);
    shared_ptr<ListNode> node2(new ListNode);
    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    node1->_next = node2;
    node2->_prev = node1;

    cout << node1.use_count() << endl;
    cout << node2.use_count() << endl;

    return 0;
}
/*
1
1
2
2
循环引用分析:
1、node1和node2两个智能指针对象指向两个节点，引用计数变成1，我们不需要手动delete。
2、node1的_next指向node2，node2的_prev指向node1，引用计数变成2。
3、node1和node2析构，引用计数减到1，所以底层的ListNode对象仍然存在。
node1管理的ListNode对象保存_next，node2管理的对象保存_prev
4、以上_next和_prev析构才能将引用计数减为0
5、但是_next属于node的成员，node1释放了，_next才会析构，而node1由_prev管理，_prev属于node2成员，所以这就叫循环引用，谁也不会释放
*/
```

将上述next和prev改为weak_ptr就行了。

unique_ptr

- 实现独占式，保证同一时间只有一个智能指针可以指向该对象。

```
// 特有操作
unique_ptr<T,D>u; // u会使用一个类型为D的可调用对象来释放它的指针
unique_ptr<T,D>u(d); // 空unique_ptr 指向类型T的对象 用类型D的对象d来代替delete
u = nullptr; // 释放u指向对象 将u置为空
u.release(); // u放弃对指针控制权，返回指针，将u置空
u.reset(); // 释放u指向对象
u.reset(q); // 释放对象，令u的内置指针为q ，否则置为空
u.reset(nullptr);
```

- 定义一个**unique_ptr**，必须将其绑定到一个**new**返回的指针上（必须直接初始化），因为没有类似**make_shared**的函数。
- 因为一个unique_ptr拥有它指向的对象，所以不允许普通拷贝和赋值。
- 但通过release或者reset可以将指针所有权从一个unique_ptr转移到另一个unique_ptr

```
unique_ptr<string>p2(p1.release());
unique_ptr<string>p3(new string("xf"));
p2.reset(p3.release());
```

向**unique_ptr**传递删除器

- 注意传递的删除器对象一般是函数指针，所以类型一般是函数指针类型，decltype(函数名)*。
- decltype作用于某个函数，它返回函数类型而不是指针类型。

5、说说你了解的**auto_ptr**作用

回答：

- 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
- 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
- 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

- 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
- 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、使用智能指针管理内存资源，**RAII**是咋回事？

回答：

RAII意为资源获取即初始化，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。

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。

7、说一说你了解的关于**lambda**的函数

回答：

谓词

是一个可调用的表达式，其返回结果是一个能用作条件的值。

一元谓词（接受单一参数）和二元谓词（接受两个参数）。接受谓词参数的算法对输入序列的元素调用谓词。元素类型必须能转换为谓词的参数类型。

我们可以向一个算法传递任何类别的可调用对象。对于一个对象或一个表达式，如果可以对其使用调用运算符（），则称它为可调用对象。

可调用对象有函数、函数指针、重载了调用运算符的类以及**lambda**表达式

一个lambda表达式表示一个可调用单元。我们可以将其理解为一个未命名的内联函数。可以定义在函数内部。

```
[捕获列表](参数列表)-> 返回类型 {函数体}
```

捕获列表是**lambda**所在函数中定义的局部变量的列表，其他三块和普通函数一样。

可以忽略参数列表和返回类型，但必须永远包含捕获列表和函数体

```
auto f = [] {return 42;}// 定义可调用对象f，不接受参数，返回42
cout << f() << endl; // 42
```

> *lambda*不能有默认参数。

可以使用lambda来调用stable_sort：

```
stable_sort(words.begin(),words.end(),[](const string &a, const
string &b) {return a.size() < b.size();});
```

使用捕获列表

如果编写一个可以传递给find_if的可调用表达式，希望这个表达式能将输入序列的每个string长度与一个固定值sz比较。find_if只能接受一元谓词，这时候就需要捕获列表了。

```
[sz](const string &a) { return a.size() >= sz;}
```

这里lambda表达式捕获了sz，并且只要单一的参数string，仍然是一元谓词，可以被find_if接受使用！

lambda捕获和返回

当定义一个**lambda**时，编译器将该表达式翻译成一个未命名类的未命名对象。未命名类含有一个重载的函数调用运算符。

例如对于传递给stable_sort作为最后一个实参的lambda表达式来说：

```
stable_sort(words.begin(), words.end(), [](const string &a, const
string &b) { return a.size() < b.size();});
```

其行为类似于下面这个类的一个未命名对象：

```
class shortstring {
public:
    bool operator()(const string &a, const string &b) const{
return a.size() < b.size();
    }
};
```

```
stable_sort(words.begin(), words.end(), shortstring()); // 第三个参数是shortstring类的对象
```

lambda其实就是函数对象（伪函数）。

所以你用auto定义一个用lambda初始化的变量，其实是定义了一个从lambda生成的类型的对象。

值捕获

采用值捕获的前提是变量可以拷贝。并且于参数不同，被捕获的变量的值是在lambda创建时拷贝，而不是调用时拷贝

```
void fcn1(){
    size_t v1 = 42;
    auto f = [v1]{ return v1;}
    v1 = 0;
    auto j = f(); // j = 42 因为创建时拷贝，而不是现在调用时。
}
```

引用捕获

```
void fcn1(){
    size_t v1 = 42;
    auto f = [&v1]{ return v1;}
    v1 = 0;
    auto j = f(); // j = 0
}
```

函数可以直接返回一个可调用对象。如果函数返回一个lambda，此lambda不能包含引用捕获，就像函数不能返回一个局部变量的引用一样。

当以引用方式捕获一个变量时，必须保证在**lambda**执行时变量是存在的

隐式捕获

除了显式列出我们希望使用的来自所在函数的变量之外，还可以让编译器根据lambda体中的代码来推断我们要使用哪些变量。

为了指示编译器推断捕获列表，应在捕获列表写一个&或=，&告诉编译器采用捕获引用方式，=则表示采用值捕获方式。

```
// sz隐式捕获，值捕获方式
wc = find_if(words.begin(),words.end(),[=](const string &s){return
s.size()>=sz;})
```

如果希望对一部分变量引用捕获，一部分值捕获。混合使用隐式捕获和显式捕获

```
// os隐式捕获，引用捕获方式  c显示捕获,值捕获
for_each(words.begin(),words.end(),[&,c](const string &s){os << s
<< c;})
```

混合使用隐式捕获和显式捕获时，捕获列表中的第一个元素必须是一个&或=。且显式捕获的变量必须使用和隐式捕获不同的方式。你是&我就值捕获。

可变**lambda**

默认情况下，对于一个值被拷贝的变量，lambda不会改变其值。如果我们希望能改变一个被捕获的变量的值，就必须在参数列表首加关键字mutable。引用的话只要它绑定的值不是const就能改变。

```
void f(){
    size_t v = 10;
    auto f = [v]()mutable{ return ++v;};
    v = 0;
    auto j = f(); // j = 11;
}
```

指定**lambda**返回类型

默认情况，如果一个lambda函数体包含return以外任何语句，编译器假定此lambda返回void。与其他返回void的函数类似，被推断返回void的lambda不能返回值。

```
[](int i){ return i < 0 ? -i : i; } // 只要return一条语句 正确
[](int i){ if (i < 0) return -i; else return i; } // 不止return一条语句 编译器推断void返回类型，但返回了int值，出错
```

所以需要使用尾置返回类型

```
[](int i)->int { if (i < 0) return -i; else return i; }
```

8、右值引用、移动构造函数

回答：

当一个对象被用作右值的时候，用的是对象的值（内容）。当对象被用作左值的时候，用的是对象的身份（在内存中的位置）。

类似一张能写字的纸，左值指这张纸本身，而右值指纸上写的字。，左值的本质是一个内存空间中的位置，而右值是一个在内存空间的某个位置存储的值。故而，一个左值可以被&取地址，可以被=赋值，而右值不能。

右值引用

- 必须绑定到右值的引用，通过&&而不是&来获得右值引用。
- 右值引用只能绑定到一个将要销毁的对象
- 一个左值表达式表示的是一个对象的身份，而一个右值表达式的是对象的值。
- 一个右值引用也不过是某个对象的另一个名字。
- 常规引用（左值引用），不能绑定到要求转换的表达式，字面常量或是返回右值的表达式。
- 右值引用则完全相反，可以将一个右值引用绑定到这类表达式上，但不能将一个右值引用直接绑定到一个左值上。

```
int i = 42;
int &r = i;                     // 正确 r引用i
int &&rr = i;                   // 错误，不能将右值引用绑定一个左值
int &r2 = i * 42;               // 错误，i * 42 是右值
const int &r3 = i * 42;         // 正确，常量引用能绑定右值
int &&rr2 = i * 42;             // 正确
```

- 返回左值引用的函数，连同赋值、下标、解引用和前置递增递减运算符，都是生成左值的例子，可以将左值引用绑定到这类表达式是上。

- 返回非引用类型的函数，连同算术、关系、位以及后置递增/递减（因为前置运算符是返回引用的）运算符都是生成右值，不能将左值引用与之绑定，可以将一个**const**左值引用或者右值引用绑定。
- 左值持久，右值短暂。右值要么是字面常量，要么是在表达式求值过程中创建的临时对象。
- 右值引用只能绑定到临时对象：
    - 所引用的对象将要销毁
    - 该对象没有其他用户
- 使用右值引用的代码可以自由地接管所引用的对象的资源。

变量是左值

- 变量都是左值。不能右值引用绑定一个右值引用类型的变量上。

```
int &&rr1 = 42;int &&rr2 = rr1;  // 错误 rr1为左值
```

标准库**move**函数

- 可以显示地将一个左值转换成为对应的右值引用类型。调用move函数来获得绑定到左值上的右值引用。

```
int &&rr3 = std::move(rr1); // ok
```

- move告诉编译器：有一个左值，我们希望像一个右值一样处理它。调用move意味着承诺：除了对rr1赋值或销毁它外，不会使用它。不能对这个移后源对象有任何假设。

移动构造函数和移动赋值运算符

- 移动构造函数的第一参数是该类类型的一个右值引用。其余额外参数应该都必须都有默认实参。
- 除了完成资源移动，移动构造函数还必须确保移后源对象处于这样一个状态——销毁它是无害的。一旦资源移动完成，源对象必须不再指向被移动的资源——这些资源的所有权归属新创建的对象。
- StrVec例子

```
StrVec::Strvec(Strvec &&s) noexcept // 移动操作不应抛出任何异常
:elements(s.elements),first_free(s.first_free),cap(s.cap)
{
    s.elements = s.first_free = s.cap = nullptr;
}
```

- 移动构造函数不分配任何新内存，它接管给定的StrVec中的内存。
- 当编写一个不抛出异常的移动操作时，我们应该将此事通知标准库。不然它会认为移动我们的类对象时可能会抛出异常，并为处理这种可能做一些额外工作。所以加一个noexcept。
- vector保证，调用push_back发生异常自身不会改变。调用push_back可能要求vector重新分配内存空间，当重新分配时，vector将元素从就空间移动到新内存中。如果重新分配过程中使用移动构造函数，且在移动了部分而不是全部元素后抛出了一个异常，就会产生问题，旧空间中的移动源元素已经改变了，而新空间中未构造的元素尚不存在。vector不能保证自身不变。但如果是拷贝构造，很简单，把新分配的内存释放了就行了。所以为避免这种情况一般用拷贝，除非你加noexcept说我的移动构造肯定不会异常，才可以使用。
- 移动赋值运算符

```
StrVec &StrVec::operator=(StrVec &&rhs) noexcept
{
    if (this != &rhs) {
        free();                    // 释放已有资源
elements = rhs.elements;
        first_free = rhs.first_free;
        cap = rhs.cap;
        rhs.elements = rhs.first_free = rhs.cap = nullptr;
    }
    return *this;
}
```

移后源对象必须可析构

合成的移动操作

> C++中的拷贝控制成员包括：
>
> 1. 构造函数
> 2. 拷贝构造函数
> 3. 拷贝赋值运算符
> 4. 移动构造函数
> 5. 移动赋值运算符
> 6. 析构函数

- 如果一个类定义了自己的拷贝构造函数、拷贝赋值运算符或者析构函数，编译器就不会为它合成移动构造函数和移动赋值运算符了。

- 只有当一个类没有定义任何自己版本的拷贝控制成员，且类的每个非static数据成员都可以移动，编译器才会为其合成移动构造函数或移动赋值运算符。编译器可以移动内置类型的成员，如果一个成员是类类型，且该类有对应移动操作，编译器也能移动。
- 移动构造函数被定义为删除的函数的条件是：
    - 有类成员定义了自己的拷贝构造函数且未定义移动构造函数，或者是有类成员未定义自己的拷贝构造函数且编译器不能为其合成移动构造函数。移动赋值运算符类似。
    - 有类成员的移动构造函数或移动赋值运算符被定义为删除的或是不可访问
    - 如果类的析构函数被定义为删除的或不可访问，则移动构造函数被定义为删除的
    - 如果有类成员是const或是引用，则类的移动赋值运算符被定义为删除的（新值赋给一个const不对的，赋给引用只是改变了引用绑定对象的值，没有给一个新对象）。
- 如果一个类定义了移动构造和/或移动赋值运算符，则类的合成拷贝构造和拷贝赋值会被定义为删除的。

如果一个类既有移动构造函数又有拷贝构造函数，则移动右值，拷贝左值。

如果没有移动构造函数，右值也被拷贝

右值引用和成员函数

- 我们希望在自己的类中强制左侧运算对象是左值，在参数列表后放一个引用限定符（指出this指向对象的左右值属性）

```
class Foo {
public:
    Foo &operator=(const Foo&) &;
    // 只能向可修改的左值赋值
};
Foo &Foo::operator=(const Foo&rhs) & {
    // 执行将rhs赋予本对象的工作
    return *this;
}
```

  引用限定符可以是&或者&&，分别指出this可以指向一个左值或右值。引用限定符只能用于（非static）成员函数且必须同时出现在声明和定义中。

  一个函数可以同时使用const和引用限定。引用限定必须跟随在const后面。

```
class Foo {
    public: Foo someMem() & const; // 错误
    Foo xf() const &; // 正确
};
```

- 可以综合引用限定符和const来区分一个成员函数的重载

```
class Foo {
public:
    Foo sorted() &&;
    Foo sorted() const &;
private:
    vector<int>data;
};
Foo Foo::sorted() &&
// Foo&& *const this ,this指向的对象是一个右值 所以可以直接对象里排
{
    sort(data.begin(),data.end());
    return *this;
}
Foo Foo::sorted() const &
// const Foo& *const this this指向的对象是一个const左值，不能直接
改变
{
    Foo ret(*this);
sort(ret.data.begin(),ret.data.end());
    return ret;
}
```

对象是一个右值，意味没有其他用户，可以改变对象。

- 如果一个成员函数有引用限定符，具有相同参数列表的所有版本都必须有引用限定符（&&或&）

9、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。

- C++11中所有的值必定属于左值和右值两者之一，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（prvalue，Pure Rvalue）或纯右值（xvalue，eXpiring Value））。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。
- 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。
- 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。

右值引用的特定：

- 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。

```
#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 发生了引用折叠
    return 0;
}
```

三、内存管理

1、类的对象存储空间？

回答：

- 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、类对象的大小受哪些因素的影响？

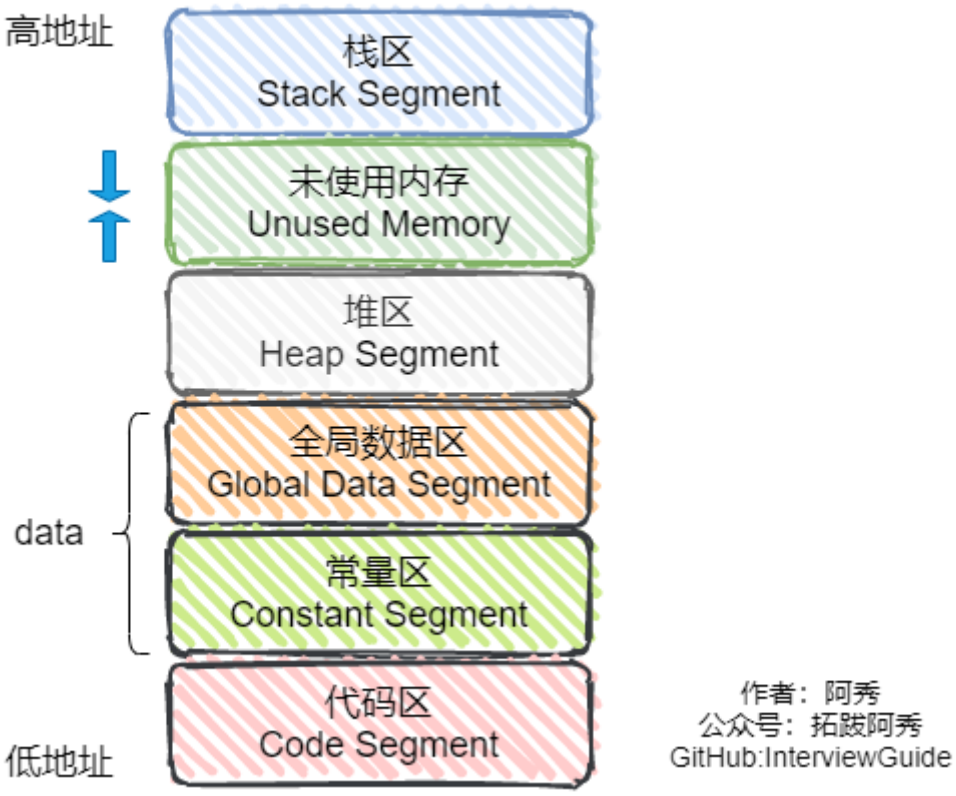
回答：

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。

3、简要说明C++的内存分区

回答：

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：如果说堆是操作系统维护的一块内存，那么自由存储区就是C++中通过new和delete动态分配和释放对象的抽象概念。需要注意的是，自由存储区和堆比较像，但不等价。

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

4、什么是内存池，如何实现？

回答：

内存池是一种内存分配的方式。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。内存池则是真正使用内存之前，先申请分配一定数量的、大小相等的内存块留作备用。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。

实现参考《**STL**源码剖析》中的内存池实现机制，之后会说。

5、可以说一下你了解的**C++**的内存管理吗？

回答：

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

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：

```
#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。

7、关于**this**指针你知道什么？

回答：

- 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使用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。

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，传入参数为当前对象地址，成员函数第一个参数为为**T * const this**

如：

```
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：

```
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。

this指针的存放

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。

8、内存泄漏？解决方法？

回答：

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) 解决方法

智能指针。

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？

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？

回答：

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、在类的析构函数中调用**delete this**，会发生什么？

回答：

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、空类大小多少？

回答：

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。
5. 要确保两个不一样的对象拥有不同的地址，就用那1byte来在内存中占用不同地址了！

13、请说一下以下几种情况下，下面几个类的大小各是多少？

回答：

```
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。

```
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

```
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

```
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;
}
```

14、this指针调用成员变量时，堆栈会发生什么变化？

回答：

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作为隐含参数传递给函数，这个隐含参数就是this指针。

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。