1 变量的声明和定义有什么区别

变量的定义为变量分配地址和存储空间， 变量的声明不分配地址。一个变量可以在多个地方声明， 但是只在一个地方定义。 加入extern 修饰的是变量的声明，说明此变量将在文件以外或在文件后面部分定义。

说明：很多时候一个变量，只是声明不分配内存空间，直到具体使用时才初始化，分配内存空间， 如外部变量。

```
int main()
{
   extern int A;
   //这是个声明而不是定义，声明A是一个已经定义了的外部变量
   //注意：声明外部变量时可以把变量类型去掉如：extern A;
   dosth(); //执行函数
}
int A; //是定义，定义了A为整型的外部变量
```

2 简述#ifdef、#else、#endif和#ifndef的作用

- 利用#ifdef、#endif将某程序功能模块包括进去，以向特定用户提供该功能。在不需要时用户可轻易将其屏蔽。

```
#ifdef MATH
#include "math.c"
#endif
```

- 在子程序前加上标记，以便于追踪和调试。

```
#ifdef DEBUG
printf ("Indebugging......!");
#endif
```

- 应对硬件的限制。由于一些具体应用环境的硬件不一样，限于条件，本地缺乏这种设备，只能绕过硬件，直接写出预期结果。

注意：虽然不用条件编译命令而直接用if语句也能达到要求，但那样做目标程序长（因为所有语句都编译），运行时间长（因为在程序运行时间对if语句进行测试）。

而采用条件编译，可以减少被编译的语句，从而减少目标程序的长度，减少运行时间。

3 写出int 、bool、 float 、指针变量与 “零值”比较的if 语句

```
//int与零值比较
if ( n == 0 )
if ( n != 0 )

//bool与零值比较
if   (flag) //   表示flag为真
if   (!flag) //   表示flag为假

//float与零值比较
const float EPSINON = 0.00001;
if ((x >= - EPSINON) && (x <= EPSINON) //其中EPSINON是允许的误差（即精度）。

//指针变量与零值比较
if (p == NULL)
if (p != NULL)
```

4 结构体可以直接赋值吗

声明时可以直接初始化，同一结构体的不同对象之间也可以直接赋值，但是当结构体中含有指针“成员”时一定要小心。

注意：当有多个指针指向同一段内存时，某个指针释放这段内存可能会导致其他指针的非法操作。因此在释放前一定要确保其他指针不再使用这段内存空间。

5 sizeof 和strlen 的区别

- sizeof是一个操作符，strlen是库函数。
- sizeof的参数可以是数据的类型，也可以是变量，而strlen只能以结尾为 ‘\0’ 的字符串作参数。
- 编译器在编译时就计算出了sizeof的结果，而strlen函数必须在运行时才能计算出来。并且sizeof计算的是数据类型占内存的大小，而strlen计算的是字符串实际的长度。
- 数组做sizeof的参数不退化，传递给strlen就退化为指针了

6 C 语言的关键字 static 和 C++ 的关键字 static 有什么区别

在 C 中 static 用来修饰局部静态变量和外部静态变量、函数。而 C++中除了上述功能外，还用来定义类的成员变量和函数。即静态成员和静态成员函数。

注意：编程时 static 的记忆性，和全局性的特点可以让在不同时期调用的函数进行通信，传递信息，而 C++的静态成员则可以在多个对象实例间进行通信，传递信息。

7 C 语言的 malloc 和 C++ 中的 new 有什么区别

- new 、delete 是操作符，可以重载，只能在C++ 中使用。
- malloc、free 是函数，可以覆盖，C、C++ 中都可以使用。
- new 可以调用对象的构造函数，对应的delete 调用相应的析构函数。
- malloc 仅仅分配内存，free 仅仅回收内存，并不执行构造和析构函数
- new 、delete 返回的是某种数据类型指针，malloc、free 返回的是void 指针。

注意：malloc 申请的内存空间要用free 释放，而new 申请的内存空间要用 delete 释放，不要混用。

8 写一个 “标准”宏MIN

```
#define min(a,b)((a)<=(b)?(a):(b))
```

9 ++i和i++的区别

++i先自增1，再返回，i++先返回i,再自增1

10 volatile有什么作用

- 状态寄存器一类的并行设备硬件寄存器。
- 一个中断服务子程序会访问到的非自动变量。
- 多线程间被几个任务共享的变量。

注意：虽然volatile在嵌入式方面应用比较多，但是在PC软件的多线程中，volatile修饰的临界变量也是非常实用的。

11 一个参数可以既是const又是volatile吗

可以，用const和volatile同时修饰变量，表示这个变量在程序内部是只读的，不能改变的，只在程序外部条件变化下改变，并且编译器不会优化这个变量。每次使用这个变量时，都要小心地去内存读取这个变量的值，而不是去寄存器读取它的备份。

注意：在此一定要注意const的意思，const只是不允许程序中的代码改变某一变量，其在编译期发挥作用，它并没有实际地禁止某段内存的读写特性。

12 a 和&a 有什么区别

&a：其含义就是“变量a的地址”。

*a：用在不同的地方，含义也不一样。

- 在声明语句中，*a只说明a是一个指针变量，如int *a;
- 在其他语句中，*a前面没有操作数且a是一个指针时，*a代表指针a指向的地址内存放的数据，如b=*a;
- *a前面有操作数且a是一个普通变量时，a代表乘以a，如c = b a。

13 用C 编写一个死循环程序

```
while(1)
{ }
```

注意：很多种途径都可实现同一种功能，但是不同的方法时间和空间占用度不同，特别是对于嵌入 式软件，处理器速度比较慢，存储空间较小，所以时间和空间优势是选择各种方法的首要考虑条件。

14 结构体内存对齐问题

请写出以下代码的输出结果：

```
#include<stdio.h>
struct S1
{
    int i:8;
    char j:4;
    int a:4;
    double b;
};

struct S2
{
    int i:8;
    char j:4;
    double b;
    int a:4;
};
```

```
struct S3
{
    int i;
    char j;
    double b;
    int a;
};

int main()
{
    printf("%d\n",sizeof(S1));  // 输出8
    printf("%d\n",sizeof(S1);  // 输出12
    printf("%d\n",sizeof(Test3));  // 输出8
    return 0;
}
```

```
sizeof(S1)=16
sizeof(S2)=24
sizeof(S3)=32
```

说明：结构体作为一种复合数据类型，其构成元素既可以是基本数据类型的变量，也可以是一些复合型类型数据。对此，编译器会自动进行成员变量的对齐以提高运算效率。默认情况下，按自然对齐条件分配空间。各个成员按照它们被声明的顺序在内存中顺序存储，第一个成员的地址和整个结构的地址相同，向结构体成员中size最大的成员对齐。

许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k（通常它为4或8）的倍数，而这个k则被称为该数据类型的对齐模数。

15 全局变量和局部变量有什么区别？实怎么实现的？操作系统和编译器是怎么知道的?

- 全局变量是整个程序都可访问的变量，谁都可以访问，生存期在整个程序从运行到结束（在程序结束时所占内存释放）；
- 而局部变量存在于模块（子程序，函数）中，只有所在模块可以访问，其他模块不可直接访问，模块结束（函数调用完毕），局部变量消失，所占据的内存释放。
- 操作系统和编译器，可能是通过内存分配的位置来知道的，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。

16 简述C、C++程序编译的内存分配情况

- 从静态存储区域分配：

内存在程序编译时就已经分配好，这块内存在程序的整个运行期间都存在。速度快、不容易出错， 因为有系统会善后。例如全局变量，static 变量，常量字符串等。

- 在栈上分配：

在执行函数时，函数内局部变量的存储单元都在栈上创建，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。大小为2M。

- 从堆上分配：

即动态内存分配。程序在运行的时候用 malloc 或new 申请任意大小的内存，程序员自己负责在何 时用free 或delete 释放内存。动态内存的生存期由程序员决定，使用非常灵活。如果在堆上分配了空间，就有责任回收它，否则运行的程序会出现内存泄漏，另外频繁地分配和释放不同大小的堆空间将会产生 堆内碎块。

一个C、C++程序编译时内存分为5 大存储区：堆区、栈区、全局区、文字常量区、程序代码区。

17 简述strcpy、sprintf 与memcpy 的区别

- 操作对象不同，strcpy 的两个操作对象均为字符串，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，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。
- 执行效率不同，memcpy 最高，strcpy 次之，sprintf 的效率最低。
- 实现功能不同，strcpy 主要实现字符串变量间的拷贝，sprintf 主要实现其他数据类型格式到字 符串的转化，memcpy 主要是内存块间的拷贝。

注意：strcpy、sprintf 与memcpy 都可以实现拷贝的功能，但是针对的对象不同，根据实际需求，来 选择合适的函数实现拷贝功能。

18 请解析(*(void (*)( ))0)( )的含义

- `void (*0)( ) ：` 是一个返回值为void，参数为空的函数指针0。
- `(void (*)( ))0：` 把0转变成一个返回值为void，参数为空的函数指针。
- `*(void (*)( ))0：` 在上句的基础上加*表示整个是一个返回值为void，无参数，并且起始地址为0的函数的名字。
- `(*(void (*)( ))0)( )：` 这就是上句的函数名所对应的函数的调用。

19 C语言的指针和引用和c++的有什么区别?

- 指针有自己的一块空间，而引用只是一个别名；

- 使用sizeof看一个指针的大小是4，而引用则是被引用对象的大小；
- 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；
- 可以有const指针，但是没有const引用；
- 指针在使用中可以指向其它对象，但是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），而引用止于一级；
- 指针和引用使用++运算符的意义不一样；
- 如果返回动态内存分配的对象或者内存，必须使用指针，引用可能引起内存泄露。

20 typedef 和define 有什么区别

- 用法不同：typedef 用来定义一种数据类型的别名，增强程序的可读性。define 主要用来定义 常量，以及书写复杂使用频繁的宏。
- 执行时间不同：typedef 是编译过程的一部分，有类型检查的功能。define 是宏定义，是预编译的部分，其发生在编译之前，只是简单的进行字符串的替换，不进行类型的检查。
- 作用域不同：typedef 有作用域限定。define 不受作用域约束，只要是在define 声明后的引用 都是正确的。
- 对指针的操作不同：typedef 和define 定义的指针时有很大的区别。

注意：typedef 定义是语句，因为句尾要加上分号。而define 不是语句，千万不能在句尾加分号。

21 指针常量与常量指针区别

指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针 是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。 指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。

注意：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用 函数中的不可改变特性。

22 简述队列和栈的异同

队列和栈都是线性存储结构，但是两者的插入和删除数据的操作不同，队列是“先进先出”，栈是 “后进先出”。

注意：区别栈区和堆区。堆区的存取是“顺序随意”，而栈区是“后进先出”。栈由编译器自动分 配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。堆一般由程序员 分配释放， 若程序员不释放，程序结束时可能由

OS 回收。分配方式类似于链表。 它与本题中的堆和栈是两回事。堆栈只是一种数据结构，而堆区和栈区是程序的不同内存存储区域。

23 设置地址为0x67a9 的整型变量的值为0xaa66

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa66;
```

注意：这道题就是强制类型转换的典型例子，无论在什么平台地址长度和整型数据的长度是一样的， 即一个整型数据可以强制转换成地址指针类型，只要有意义即可。

24 编码实现字符串转化为数字

编码实现函数atoi()，设计一个程序，把一个字符串转化为一个整型数值。例如数字：“5486321”， 转化成字符：5486321。

```
int myAtoi(const char * str)
{
   int num = 0; //保存转换后的数值
   int isNegative = 0; //记录字符串中是否有负号

   int n =0;
   char *p = str;
   if(p == NULL) //判断指针的合法性
   {
      return -1;
   }
   while(*p++ != '\0') //计算数字符串度
   {
      n++;
   }
   p = str;
   if(p[0] == '-') //判断数组是否有负号
   {
      isNegative = 1;
   }

   char temp = '0';
   for(int i = 0 ; i < n; i++)
   {
      char temp = *p++;
       if(temp > '9' ||temp < '0') //滤除非数字字符
      {
```

```
            continue;
        }
        if(num !=0 || temp != '0') //滤除字符串开始的0 字符
        {
            temp -= 0x30; //将数字字符转换为数值
             num += temp *int( pow(10 , n - 1 -i) );
         }
    }
    if(isNegative) //如果字符串中有负号，将数值取反
    {
        return (0 - num);
    }
    else
    {
        return num; //返回转换后的数值
    }
}
```

25 C语言的结构体和C++的有什么区别

- C语言的结构体是不能有函数成员的，而C++的类可以有。
- C语言的结构体中数据成员是没有private、public和protected访问限定的。而C++的类的成员有这些访问限定。
- C语言的结构体是没有继承关系的，而C++的类却有丰富的继承关系。

注意：虽然C的结构体和C++的类有很大的相似度，但是类是实现面向对象的基础。而结构体只可以简单地理解为类的前身。

26 简述指针常量与常量指针的区别

- 指针常量是指定义了一个指针，这个指针的值只能在定义时初始化，其他地方不能改变。常量指针是指定义了一个指针，这个指针指向一个只读的对象，不能通过常量指针来改变这个对象的值。
- 指针常量强调的是指针的不可改变性，而常量指针强调的是指针对其所指对象的不可改变性。

注意：无论是指针常量还是常量指针，其最大的用途就是作为函数的形式参数，保证实参在被调用函数中的不可改变特性。

27 如何避免“野指针”

- 指针变量声明时没有被初始化。解决办法：指针声明时初始化，可以是具体的地址值，也可让它指向NULL。
- 指针p被free或者delete之后，没有置为NULL。解决办法：指针指向的内存空间被释放后指针应该指向NULL。

- 指针操作超越了变量的作用范围。解决办法：在变量的作用域结束前释放掉变量的地址空间并且让指针指向NULL。

28 句柄和指针的区别和联系是什么?

句柄和指针其实是两个截然不同的概念。Windows系统用句柄标记系统资源，隐藏系统的信息。你只要知道有这个东西，然后去调用就行了，它是个32it的uint。指针则标记某个物理内存地址，两者是不同的概念。

29 new/delete与malloc/free的区别是什么

- new能自动计算需要分配的内存空间，而malloc需要手工计算字节数。

```
int *p = new int[2];
int *q = (int *)malloc(2*sizeof(int));
```

- new与delete直接带具体类型的指针，malloc和free返回void类型的指针。
- new类型是安全的，而malloc不是。例如int *p = new float[2];就会报错；而int p = malloc(2 sizeof(int))编译时编译器就无法指出错误来。
- new一般分为两步：new操作和构造。new操作对应与malloc，但new操作可以重载，可以自定义内存分配策略，不做内存分配，甚至分配到非内存设备上，而malloc不行。
- new调用构造函数，malloc不能；delete调用析构函数，而free不能。
- malloc/free需要库文件stdlib.h的支持，new/delete则不需要!

注意：delete和free被调用后，内存不会立即回收，指针也不会指向空，delete或free仅仅是告诉操作系统，这一块内存被释放了，可以用作其他用途。但是由于没有重新对这块内存进行写操作，所以内存中的变量数值并没有发生变化，出现野指针的情况。因此，释放完内存后，应该讲该指针指向NULL。

30 说一说extern “C”

extern "C"的主要作用就是为了能够正确实现C++代码调用其他C语言代码。加上extern "C"后，会指示编译器这部分代码按C语言（而不是C++）的方式进行编译。由于C++支持函数重载，因此编译器编译函数的过程中会将函数的参数类型也加到编译后的代码中，而不仅仅是函数名；而C语言并不支持函数重载，因此编译C语言代码的函数时不会带上函数的参数类型，一般只包括函数名。

这个功能十分有用处，因为在C++出现以前，很多代码都是C语言写的，而且很底层的库也是C语言写的，为了更好的支持原来的C代码和已经写好的C语言库，需要在C++中尽可能的支持C，而extern "C"就是其中的一个策略。

- C++代码调用C语言代码
- 在C++的头文件中使用
- 在多个人协同开发时，可能有的人比较擅长C语言，而有的人擅长C++，这样的情况下也会有用到

31 请你来说一下C++中struct和class的区别

在C++中，class和struct做类型定义是只有两点区别：

- 默认继承权限不同，class继承默认是private继承，而struct默认是public继承
- class还可用于定义模板参数，像typename，但是关键字struct不能同于定义模板参数 C++保留struct关键字，原因
- 保证与C语言的向下兼容性，C++必须提供一个struct
- C++中的struct定义必须百分百地保证与C语言中的struct的向下兼容性，把C++中的最基本的对象单元规定为class而不是struct，就是为了避免各种兼容性要求的限制
- 对struct定义的扩展使C语言的代码能够更容易的被移植到C++中

32 C++类内可以定义引用数据成员吗?

- 可以，必须通过成员函数初始化列表初始化。

33 C++中类成员的访问权限

C++通过 public、protected、private 三个关键字来控制成员变量和成员函数的访问权限，它们分别表示公有的、受保护的、私有的，被称为成员访问限定符。在类的内部（定义类的代码内部），无论成员被声明为 public、protected 还是 private，都是可以互相访问的，没有访问权限的限制。在类的外部（定义类的代码之外），只能通过对象访问成员，并且通过对象只能访问 public 属性的成员，不能访问 private、protected 属性的成员

34 什么是右值引用，跟左值又有什么区别?

左值和右值的概念：

- 左值：能取地址，或者具名对象，表达式结束后依然存在的持久对象；
- 右值：不能取地址，匿名对象，表达式结束后就不再存在的临时对象； 区别：

- 左值能寻址，右值不能；
- 左值能赋值，右值不能；
- 左值可变，右值不能（仅对基础类型适用，用户自定义类型右值引用可以通过成员函数改变）；

35 面向对象的三大特征

- 封装性：将客观事物抽象成类，每个类对自身的数据和方法实行 protection （private， protected，public ）。
- 继承性：广义的继承有三种实现形式：实现继承（使用基类的属性和方法而无需额外编码的能力)、可 视继承(子窗体使用父窗体的外观和实现代码)、接口继承(仅使用属性和方法,实现滞后到子类实现)。
- 多态性：是将父类对象设置成为和一个或更多它的子对象相等的技术。用子类对象给父类对象赋值 之后，父类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。

36 说一说c++中四种cast转换

C++中四种类型转换是：static_cast，dynamic_cast，const_cast，reinterpret_cast

1、const_cast

- 用于将const变量转为非const

2、static_cast

- 用于各种隐式转换，比如非const转const，void*转指针等, static_cast能用于多态向上转化，如果向下转能成功但是不安全，结果未知；

3、dynamic_cast

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和向下转化。只能转指针或引用。向下转化时，如果是非法的***对于指针返回NULL，对于引用抛异常***。要深入了解内部转换的原理。

- 向上转换：指的是子类向基类的转换
- 向下转换：指的是基类向子类的转换

它通过判断在执行到该语句的时候变量的运行时类型和要转换的类型是否相同来判断是否能够进行向下转换。

4、reinterpret_cast

- 几乎什么都可以转，比如将int转指针，可能会出问题，尽量少用；

5、为什么不使用C的强制转换？

- C的强制转换表面上看起来功能强大什么都能转，但是转化不够明确，不能进行错误检查，容易出错。

37 C++的空类有哪些成员函数

- 缺省构造函数。
- 缺省拷贝构造函数。
- 缺省析构函数。
- 缺省赋值运算符。
- 缺省取址运算符。
- 缺省取址运算符 const 。

注意：有些书上只是简单的介绍了前四个函数。没有提及后面这两个函数。但后面这两个函数也是 空类的默认函数。另外需要注意的是，只有当实际使用这些函数的时候，编译器才会去定义它们。

38 对c++中的smart pointer四个智能指针：shared_ptr,unique_ptr,weak_ptr,auto_ptr的理解

C++里面的四个智能指针：auto_ptr，shared_ptr，weak_ptr，unique_ptr 其中后三个是c++11支持，并且第一个已经被11弃用。

智能指针的作用是管理一个指针，因为存在以下这种情况：申请的空间在函数结束时忘记释放，造成内存泄漏。使用智能指针可以很大程度上的避免这个问题，因为智能指针就是一个类，当超出了类的作用域是，类会自动调用析构函数，析构函数会自动释放资源。所以智能指针的作用原理就是在函数结束时自动释放内存空间，不需要手动释放内存空间。

- auto_ptr（c++98的方案，cpp11已经抛弃）

采用所有权模式。

```
auto_ptr< string> p1 (new string ("I reigned lonely as a cloud.”));
auto_ptr<string> p2;
p2 = p1; //auto_ptr不会报错.
```

此时不会报错，p2剥夺了p1的所有权，但是当程序运行时访问p1将会报错。所以auto_ptr的缺点是：存在潜在的内存崩溃问题！

- unique_ptr（替换auto_ptr）

unique_ptr实现独占式拥有或严格拥有概念，保证同一时间内只有一个智能指针可以指向该对象。它对于避免资源泄露(例如“以new创建对象后因为发生异常而忘记调用delete”)特别有用。

采用所有权模式。

```
unique_ptr<string> p3 (new string ("auto"));   //#4
unique_ptr<string> p4;                         //#5
p4 = p3;//此时会报错！！
```

编译器认为p4=p3非法，避免了p3不再指向有效数据的问题。因此，unique_ptr比auto_ptr更安全。

另外unique_ptr还有更聪明的地方：当程序试图将一个 unique_ptr 赋值给另一个时，如果源 unique_ptr 是个临时右值，编译器允许这么做；如果源 unique_ptr 将存在一段时间，编译器将禁止这么做，比如：

```
unique_ptr<string> pu1(new string ("hello world"));
unique_ptr<string> pu2;
pu2 = pu1;                                     // #1 not allowed
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string ("You"));   // #2 allowed
```

其中#1留下悬挂的unique_ptr(pu1)，这可能导致危害。而#2不会留下悬挂的unique_ptr，因为它调用 unique_ptr 的构造函数，该构造函数创建的临时对象在其所有权让给 pu3 后就会被销毁。这种随情况而已的行为表明，unique_ptr优于允许两种赋值的auto_ptr 。

注：如果确实想执行类似与#1的操作，要安全的重用这种指针，可给它赋新值。C++有一个标准库函数std::move()，让你能够将一个unique_ptr赋给另一个。例如：

```
unique_ptr<string> ps1, ps2;
ps1 = demo("hello");
ps2 = move(ps1);
ps1 = demo("alexia");
cout << *ps2 << *ps1 << endl;
```

- shared_ptr

shared_ptr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。从名字share就可以看出了资源可以被多个指针共享，它使用计数机制来表明资源被几个指针共享。可以通过成员函数use_count()来查看资源的所有者个数。除了可以通过new来构造，还可以通过传入auto_ptr，unique_ptr,weak_ptr来构造。当我们调用release()时，当前指针会释放资源所有权，计数减一。当计数等于0时，资源会被释放。

shared_ptr 是为了解决 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。

成员函数：

use_count 返回引用计数的个数

unique 返回是否是独占所有权(use_count 为 1)

swap 交换两个 shared_ptr 对象(即交换所拥有的对象)

reset 放弃内部对象的所有权或拥有对象的变更，会引起原有对象的引用计数的减少

get 返回内部对象(指针)，由于已经重载了()方法，因此和直接使用对象是一样的.如 shared_ptrsp(new int(1)); sp 与 sp.get()是等价的

- weak_ptr

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。weak_ptr是用来解决shared_ptr相互引用时的死锁问题,如果说两个shared_ptr相互引用,那么这两个指针的引用计数永远不可能下降为0,资源永远不会释放。它是对对象的一种弱引用，不会增加对象的引用计数，和shared_ptr之间可以相互转化，shared_ptr可以直接赋值给它，它可以通过调用lock函数来获得shared_ptr。

```
class B;
class A
{
```

```
public:
shared_ptr<B> pb_;
~A()
{
     cout<<"A delete
";
}
};
class B
{
public:
shared_ptr<A> pa_;
~B()
{
    cout<<"B delete
";
}
};
void fun()
{
    shared_ptr<B> pb(new B());
    shared_ptr<A> pa(new A());
    pb->pa_ = pa;
    pa->pb_ = pb;
    cout<<pb.use_count()<<endl;
    cout<<pa.use_count()<<endl;
}
int main()
{
    fun();
    return 0;
}
```

可以看到fun函数中pa ，pb之间互相引用，两个资源的引用计数为2，当要跳出函数时，智能指针pa，pb析构时两个资源引用计数会减一，但是两者引用计数还是为1，导致跳出函数时资源没有被释放（A B的析构函数没有被调用），如果把其中一个改为weak_ptr就可以了，我们把类A里面的shared_ptr pb_；改为weak_ptr pb_；运行结果如下，这样的话，资源B的引用开始就只有1，当pb析构时，B的计数变为0，B得到释放，B释放的同时也会使A的计数减一，同时pa析构时使A的计数减一，那么A的计数为0，A得到释放。

注意：不能通过weak_ptr直接访问对象的方法，比如B对象中有一个方法print(),我们不能这样访问，pa->pb_->print(); 英文pb_是一个weak_ptr，应该先把它转化为shared_ptr,如：shared_ptr p = pa->pb_.lock(); p->print();

39 说说强制类型转换运算符

static_cast

- 用于非多态类型的转换
- 不执行运行时类型检查（转换安全性不如 dynamic_cast）
- 通常用于转换数值数据类型（如 float -> int）
- 可以在整个类层次结构中移动指针，子类转化为父类安全（向上转换），父类转化为子类不安全（因为子类可能有不在父类的字段或方法）

dynamic_cast

- 用于多态类型的转换
- 执行行运行时类型检查
- 只适用于指针或引用
- 对不明确的指针的转换将失败（返回 nullptr），但不引发异常
- 可以在整个类层次结构中移动指针，包括向上转换、向下转换

const_cast

- 用于删除 const、volatile 和 __unaligned 特性（如将 const int 类型转换为 int 类型） reinterpret_cast
- 用于位的简单重新解释
- 滥用 reinterpret_cast 运算符可能很容易带来风险。除非所需转换本身是低级别的，否则应- 使用其他强制转换运算符之一。
- 允许将任何指针转换为任何其他指针类型（如 char* 到 int* 或 One_class* 到 Unrelated_class* 之类的转换，但其本身并不安全）
- 也允许将任何整数类型转换为任何指针类型以及反向转换。
- reinterpret_cast 运算符不能丢掉 const、volatile 或 __unaligned 特性。
- reinterpret_cast 的一个实际用途是在哈希函数中，即，通过让两个不同的值几乎不以相同的索引结尾的方式将值映射到索引。

bad_cast

- 由于强制转换为引用类型失败，dynamic_cast 运算符引发 bad_cast 异常。

bad_cast 使用

```
try {
    Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
}
```

```
catch (bad_cast b) {
    cout << "Caught: " << b.what();
}
```

40 谈谈你对拷贝构造函数和赋值运算符的认识

拷贝构造函数和赋值运算符重载有以下两个不同之处：

- 拷贝构造函数生成新的类对象，而赋值运算符不能。
- 由于拷贝构造函数是直接构造一个新的类对象，所以在初始化这个对象之前不用检验源对象 是否和新建对象相同。而赋值运算符则需要这个操作，另外赋值运算中如果原来的对象中有内存分配要先把内存释放掉。

注意：当有类中有指针类型的成员变量时，一定要重写拷贝构造函数和赋值运算符，不要使用默认 的。

41 在C++中，使用malloc申请的内存能否通过delete释放？使用new申请的内存能否用free?

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。

42 用C++设计一个不能被继承的类

```
template <typename T> class A
{
   friend T;
    private:
     A() {}
    ~A() {}
};
class B : virtual public A<B>
{
   public:
    B() {}
   ~B() {}
};
class C : virtual public B
{
   public:
```

```
    C() {}
    ~C() {}
};
void main( void )
{
    B b;
    //C c;
    return;
}
```

注意：构造函数是继承实现的关键，每次子类对象构造时，首先调用的是父类的构造函数，然后才 是自己的。

43 C++自己实现一个String类

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

using namespace std;

class String{
public:
    // 默认构造函数
    String(const char *str = nullptr);
    // 拷贝构造函数
    String(const String &str);
    // 析构函数
    ~String();
    // 字符串赋值函数
    String& operator=(const String &str);

private:
    char *m_data;
    int m_size;
};

// 构造函数
String::String(const char *str)
{
    if(str == nullptr)  // 加分点：对m_data加NULL 判断
    {
        m_data = new char[1];   // 得分点：对空字符串自动申请存放结束标志'\0'的
        m_data[0] = '\0';
        m_size = 0;
    }
    else
    {
```

```
        m_size = strlen(str);
        m_data = new char[m_size + 1];
        strcpy(m_data, str);
    }
}

// 拷贝构造函数
String::String(const String &str)   // 得分点：输入参数为const型
{
    m_size = str.m_size;
    m_data = new char[m_size + 1];  //加分点：对m_data加NULL 判断
    strcpy(m_data, str.m_data);
}

// 析构函数
String::~String()
{
    delete[] m_data;
}

// 字符串赋值函数
String& String::operator=(const String &str)  // 得分点：输入参数为const
{
    if(this == &str)    //得分点：检查自赋值
        return *this;

    delete[] m_data;    //得分点：释放原有的内存资源
    m_size = strlen(str.m_data);
    m_data = new char[m_size + 1];  //加分点：对m_data加NULL 判断
    strcpy(m_data, str.m_data);
    return *this;       //得分点：返回本对象的引用
}
```

44 访问基类的私有虚函数

写出以下程序的输出结果：

```
#include <iostream.h>
class A
{
   virtual void g()
   {
      cout << "A::g" << endl;
   }
  private:
   virtual void f()
   {
```

```
        cout << "A::f" << endl;
    }
};
class B : public A
{
    void g()
    {
        cout << "B::g" << endl;
    }
    virtual void h()
    {
        cout << "B::h" << endl;
    }
};
typedef void( *Fun )( void );
void main()
{
    B b;
    Fun pFun;
    for(int i = 0 ; i < 3; i++)
    {
        pFun = ( Fun )*( ( int* ) * ( int* )( &b ) + i );
        pFun();
    }
}
```

输出结果：

```
B::g
A::f
B::h
```

注意：考察了面试者对虚函数的理解程度。一个对虚函数不了解的人很难正确的做出本题。 在学习面向对象的多态性时一定要深刻理解虚函数表的工作原理。

45 对虚函数和多态的理解

多态的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的，在运行期间动态绑定。举个例子：一个父类类型的指针指向一个子类对象时候，使用父类的指针去调用子类中重写了的父类中的虚函数的时候，会调用子类重写过后的函数，在父类中声明为加了virtual关键字的函数，在子类中重写时候不需要加virtual也是虚函数。

虚函数的实现：在有虚函数的类中，类的最开始部分是一个虚函数表的指针，这个指针指向一个虚函数表，表中放了虚函数的地址，实际的虚函数在代码段(.text)中。当子类继承了父类的时候也会继承其虚函数表，当子类重写父类中虚函数时候，会将其继承到的虚函数表中的地址替换为重新写的函数地址。使用了虚函数，会增加访问内存开销，降低效率。

46 简述类成员函数的重写、重载和隐藏的区别

（1）重写和重载主要有以下几点不同。

- 范围的区别：被重写的和重写的函数在两个类中，而重载和被重载的函数在同一个类中。
- 参数的区别：被重写函数和重写函数的参数列表一定相同，而被重载函数和重载函数的参数列表一 定不同。
- virtual 的区别：重写的基类中被重写的函数必须要有virtual 修饰，而重载函数和被重载函数可以被 virtual 修饰，也可以没有。

（2）隐藏和重写、重载有以下几点不同。

- 与重载的范围不同：和重写一样，隐藏函数和被隐藏函数不在同一个类中。
- 参数的区别：隐藏函数和被隐藏的函数的参数列表可以相同，也可不同，但是函数名肯定要相同。 当参数不相同时，无论基类中的参数是否被virtual 修饰，基类的函数都是被隐藏，而不是被重写。

注意：虽然重载和覆盖都是实现多态的基础，但是两者实现的技术完全不相同，达到的目的也是完 全不同的，覆盖是动态态绑定的多态，而重载是静态绑定的多态。

47 链表和数组有什么区别

- 存储形式：数组是一块连续的空间，声明时就要确定长度。链表是一块可不连续的动态空间， 长度可变，每个结点要保存相邻结点指针。
- 数据查找：数组的线性查找速度快，查找操作直接使用偏移地址。链表需要按顺序检索结点， 效率低。
- 数据插入或删除：链表可以快速插入和删除结点，而数组则可能需要大量数据移动。
- 越界问题：链表不存在越界问题，数组有越界问题。

注意：在选择数组或链表数据结构时，一定要根据实际需要进行选择。数组便于查询，链表便于插 入删除。数组节省空间但是长度固定，链表虽然变长但是占了更多的存储空间。

48 用两个栈实现一个队列的功能

```
typedef struct node
{
```

```
 int data;
 node *next;
}node,*LinkStack;

//创建空栈:
LinkStack CreateNULLStack( LinkStack &S)
{
 S = (LinkStack)malloc( sizeof( node ) ); // 申请新结点
 if( NULL == S)
 {
  printf("Fail to malloc a new node.\n");

  return NULL;
 }
 S->data = 0; //初始化新结点
 S->next = NULL;

 return S;
}

//栈的插入函数:
LinkStack Push( LinkStack &S, int data)
{
 if( NULL == S) //检验栈
 {
  printf("There no node in stack!");
  return NULL;
 }

 LinkStack p = NULL;
 p = (LinkStack)malloc( sizeof( node ) ); // 申请新结点

 if( NULL == p)
 {
  printf("Fail to malloc a new node.\n");
  return S;
 }
 if( NULL == S->next)
 {
  p->next = NULL;
 }
 else
 {
  p->next = S->next;
 }
 p->data = data; //初始化新结点
 S->next = p; //插入新结点
 return S;
}
```

```
//出栈函数：
node Pop( LinkStack &S)
{
 node temp;
 temp.data = 0;
 temp.next = NULL;

 if( NULL == S) //检验栈
 {
  printf("There no node in stack!");
  return temp;
 }
 temp = *S;

 if( S->next == NULL )
 {
  printf("The stack is NULL,can't pop!\n");
  return temp;
 }
 LinkStack p = S ->next; //节点出栈

 S->next = S->next->next;
 temp = *p;
 free( p );
 p = NULL;

 return temp;
}

//双栈实现队列的入队函数：
LinkStack StackToQueuePush( LinkStack &S, int data)
{
 node n;
 LinkStack S1 = NULL;
 CreateNULLStack( S1 ); //创建空栈

 while( NULL != S->next ) //S 出栈入S1
 {
  n = Pop( S );
  Push( S1, n.data );
 }
 Push( S1, data ); //新结点入栈

 while( NULL != S1->next ) //S1 出栈入S
 {
  n = Pop( S1 );
  Push( S, n.data );
 }
 return S;
}
```

注意：用两个栈能够实现一个队列的功能，那用两个队列能否实现一个队列的功能呢？结果是否定 的，因为栈是先进后出，将两个栈连在一起，就是先进先出。而队列是现先进先出，无论多少个连在一 起都是先进先出，而无法实现先进后出。

49 vector的底层原理

vector底层是一个动态数组，包含三个迭代器，start和finish之间是已经被使用的空间范围，end_of_storage是整块连续空间包括备用空间的尾部。

当空间不够装下数据（vec.push_back(val)）时，会自动申请另一片更大的空间（1.5倍或者2倍），然后把原来的数据拷贝到新的内存空间，接着释放原来的那片空间[vector内存增长机制]。

当释放或者删除（vec.clear()）里面的数据时，其存储空间不释放，仅仅是清空了里面的数据。因此，对vector的任何操作一旦引起了空间的重新配置，指向原vector的所有迭代器会都失效了。

50 vector中的reserve和resize的区别

- reserve是直接扩充到已经确定的大小，可以减少多次开辟、释放空间的问题（优化push_back），就可以提高效率，其次还可以减少多次要拷贝数据的问题。reserve只是保证vector中的空间大小（capacity）最少达到参数所指定的大小n。reserve()只有一个参数。
- resize()可以改变有效空间的大小，也有改变默认值的功能。capacity的大小也会随着改变。resize()可以有多个参数。

51 vector中的size和capacity的区别

- size表示当前vector中有多少个元素（finish - start）；
- capacity函数则表示它已经分配的内存中可以容纳多少元素（end_of_storage - start）；

52 vector中erase方法与algorithn中的remove方法区别

- vector中erase方法真正删除了元素，迭代器不能访问了
- remove只是简单地将元素移到了容器的最后面，迭代器还是可以访问到。因为algorithm通过迭代器进行操作，不知道容器的内部结构，所以无法进行真正的删除。

53 vector迭代器失效的情况

- 当插入一个元素到vector中，由于引起了内存重新分配，所以指向原内存的迭代器全部失效。
- 当删除容器中一个元素后,该迭代器所指向的元素已经被删除，那么也造成迭代器失效。erase方法会返回下一个有效的迭代器，所以当我们要删除某个元素时，需要it=vec.erase(it);。

54 正确释放vector的内存(clear(), swap(), shrink_to_fit())

- vec.clear()：清空内容，但是不释放内存。
- vector().swap(vec)：清空内容，且释放内存，想得到一个全新的vector。
- vec.shrink_to_fit()：请求容器降低其capacity和size匹配。
- vec.clear();vec.shrink_to_fit();：清空内容，且释放内存。

55 list的底层原理

- ist的底层是一个双向链表，使用链表存储数据，并不会将它们存储到一整块连续的内存空间中。恰恰相反，各元素占用的存储空间（又称为节点）是独立的、分散的，它们之间的线性关系通过指针来维持,每次插入或删除一个元素，就配置或释放一个元素空间。
- list不支持随机存取，如果需要大量的插入和删除，而不关心随即存取

56 什么情况下用vector，什么情况下用list，什么情况下用deque

- vector可以随机存储元素（即可以通过公式直接计算出元素地址，而不需要挨个查找），但在非尾部插入删除数据时，效率很低，适合对象简单，对象数量变化不大，随机访问频繁。除非必要，我们尽可能选择使用vector而非deque，因为deque的迭代器比vector迭代器复杂很多。
- list不支持随机存储，适用于对象大，对象数量变化频繁，插入和删除频繁，比如写多读少的场景。
- 需要从首尾两端进行插入或删除操作的时候需要选择deque。

57 priority_queue的底层原理

- priority_queue：优先队列，其底层是用堆来实现的。在优先队列中，队首元素一定是当前队列中优先级最高的那一个。

58 map 、set、multiset、multimap的底层原理

map 、set、multiset、multimap的底层实现都是红黑树，epoll模型的底层数据结构也是红黑树，linux系统中CFS进程调度算法，也用到红黑树。

红黑树的特性：

- 每个结点或是红色或是黑色；
- 根结点是黑色；
- 每个叶结点是黑的；
- 如果一个结点是红的，则它的两个儿子均是黑色；
- 每个结点到其子孙结点的所有路径上包含相同数目的黑色结点。

59 为何map和set的插入删除效率比其他序列容器高

- 因为不需要内存拷贝和内存移动

60 为何map和set每次Insert之后，以前保存的iterator不会失效?

- 因为插入操作只是结点指针换来换去，结点内存没有改变。而iterator就像指向结点的指针，内存没变，指向内存的指针也不会变。

61 当数据元素增多时（从10000到20000），map的set的查找速度会怎样变化?

- RB-TREE用二分查找法，时间复杂度为logn，所以从10000增到20000时，查找次数从log10000=14次到log20000=15次，多了1次而已。

62 map 、set、multiset、multimap的特点

- set和multiset会根据特定的排序准则自动将元素排序，set中元素不允许重复，multiset可以重复。
- map和multimap将key和value组成的pair作为元素，根据key的排序准则自动将元素排序（因为红黑树也是二叉搜索树，所以map默认是按key排序的），map中元素的key不允许重复，multimap可以重复。
- map和set的增删改查速度为都是logn，是比较高效的。

63 为何map和set的插入删除效率比其他序列容器高，而且每次insert之后，以前保存的iterator不会失效?

- 存储的是结点，不需要内存拷贝和内存移动。
- 插入操作只是结点指针换来换去，结点内存没有改变。而iterator就像指向结点的指针，内存没变，指向内存的指针也不会变。

64 为何map和set不能像vector一样有个reserve函数来预分配数据?

- 在map和set内部存储的已经不是元素本身了，而是包含元素的结点。也就是说map内部使用的Alloc并不是map<Key, Data, Compare, Alloc>声明的时候从参数中传入的Alloc。

65 set的底层实现实现为什么不用哈希表而使用红黑树?

- set中元素是经过排序的，红黑树也是有序的，哈希是无序的
- 如果只是单纯的查找元素的话，那么肯定要选哈希表了，因为哈希表在的最好查找时间复杂度为O(1)，并且如果用到set中那么查找时间复杂度的一直是O（1），因为set中是不允许有元素重复的。而红黑树的查找时间复杂度为O(lgn)

66 hash_map与map的区别？什么时候用hash_map，什么时候用map?

- 构造函数：hash_map需要hash function和等于函数，而map需要比较函数（大于或小于）。
- 存储结构：hash_map以hashtable为底层，而map以RB-TREE为底层。

- 总的说来，hash_map查找速度比map快，而且查找速度基本和数据量大小无关，属于常数级别。而map的查找速度是logn级别。但不一定常数就比log小，而且hash_map还有hash function耗时。
- 如果考虑效率，特别当元素达到一定数量级时，用hash_map。
- 考虑内存，或者元素数量较少时，用map。

67 迭代器失效的问题

插入操作：

- 对于vector和string，如果容器内存被重新分配，iterators,pointers,references失效；如果没有重新分配，那么插入点之前的iterator有效，插入点之后的iterator失效；
- 对于deque，如果插入点位于除front和back的其它位置，iterators,pointers,references失效；当我们插入元素到front和back时，deque的迭代器失效，但reference和pointers有效；
- 对于list和forward_list，所有的iterator,pointer和refercnce有效。 删除操作：
- 对于vector和string，删除点之前的iterators,pointers,references有效；off-the-end迭代器总是失效的；
- 对于deque，如果删除点位于除front和back的其它位置，iterators,pointers,references失效；当我们插入元素到front和back时，off-the-end失效，其他的iterators,pointers,references有效；
- 对于list和forward_list，所有的iterator,pointer和refercnce有效。
- 对于关联容器map来说，如果某一个元素已经被删除，那么其对应的迭代器就失效了，不应该再被使用，否则会导致程序无定义的行为。

68 STL线程不安全的情况

- 在对同一个容器进行多线程的读写、写操作时；
- 在每次调用容器的成员函数期间都要锁定该容器；
- 在每个容器返回的迭代器（例如通过调用begin或end）的生存期之内都要锁定该容器；
- 在每个在容器上调用的算法执行期间锁定该容器。