C++常见面试题

一些语法题

1、new、delete、malloc、free关系

delete会调用对象的析构函数,和new对应；free只会释放内存，new调用构造函数。malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。对于非内部数据类型的对象而言，光用maloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。由于malloc/free是库函数而不是运算符，不在编译器控制权限之内，不能够把执行构造函数和析构函数的任务强加于malloc/free。因此C++语言需要一个能完成动态内存分配和初始化工作的运算符new，以及一个能完成清理与释放内存工作的运算符delete。注意new/delete不是库函数。

2、delete与delete[] 区别

delete只会调用一次析构函数，而delete[]会调用每一个成员的析构函数。在More Effective C++中有更为详细的解释："当delete操作符用于数组时，它为每个数组元素调用析构函数，然后调用operator delete来释放内存。"delete与new配套，delete []与new []配套

```
MemTest *mTest1=new MemTest[10];
MemTest *mTest2=new MemTest;
Int *pInt1=new int [10];
Int *pInt2=new int;
delete[]pInt1; //-1-
delete[]pInt2; //-2-
delete[]mTest1;//-3-
delete[]mTest2;//-4-

```

在-4-处报错。

这就说明：对于内建简单数据类型，delete和delete[]功能是相同的。对于自定义的复杂数据类型，delete和delete[]不能互用。delete[]删除一个数组，delete删除一个指针。简单来说，用new分配的内存用delete删除；用new[]分配的内存用delete[]删除。delete[]会调用数组元素的析构函数。内部数据类型没有析构函数，所以问题不大。如果你在用delete时没用括号，delete就会认为指向的是单个对象，否则，它就会认为指向的是一个数组。

3、C++有哪些性质（面向对象的特点）

封装，继承和多态。

4、子类析构时要调用父类的析构函数吗?

析构函数调用的次序是先派生类的析构后基类的析构，也就是说在基类的的析构调用的时候,派生类的信息已经全部销毁了。定义一个对象时先调用基类的构造函数、然后调用派生类的构造函数；析构的时候恰好相反：先调用派生类的析构函数、然后调用基类的析构函数。

5、多态，虚函数，纯虚函数

多态：是对于不同对象接收相同消息时产生不同的动作。C++的多态性具体体现在运行和编译两个方面：在程序运行时的多态性通过继承和虚函数来体现；

在程序编译时多态性体现在函数和运算符的重载上；

虚函数：在基类中冠以关键字 virtual 的成员函数。 它提供了一种接口界面。允许在派生类中对基类的虚函数重新定义。

纯虚函数的作用：在基类中为其派生类保留一个函数的名字，以便派生类根据需要对它进行定义。作为接口而存在纯虚函数不具备函数的功能，一般不能直接被调用。

从基类继承来的纯虚函数，在派生类中仍是虚函数。如果一个类中至少有一个纯虚函数，那么这个类被称为抽象类（abstract class）。

抽象类中不仅包括纯虚函数，也可包括虚函数。抽象类必须用作派生其他类的基类，而不能用于直接创建对象实例。但仍可使用指向抽象类的指针支持运行时多态性。

6、求下面函数的返回值（微软）

```
int func(x)
{
   int countx = 0;
   while(x)
   {
      countx ++;
      x = x&(x-1);
   }
   return countx;
}
```

假定x = 9999。 答案：8

思路：将x转化为2进制，看含有的1的个数。

7、什么是“引用”？申明和使用“引用”要注意哪些问题？

答：引用就是某个目标变量的“别名”(alias)，对应用的操作与对变量直接操作效果完全相同。申明一个引用的时候，切记要对其进行初始化。引用声明完毕后，相当于目标变量名有两个名称，即该目标原名称和引用名，不能再把该引用名作为其他变量名的别名。声明一个引用，不是新定义了一个变量，它只表示该引用名是目标变量名的一个别名，它本身不是一种数据类型，因此引用本身不占存储单元，系统也不给引用分配存储单元。不能建立数组的引用。

8、将“引用”作为函数参数有哪些特点？

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。

2、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。

3、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。

9、在什么时候需要使用“常引用”?

如果既要利用引用提高程序的效率，又要保护传递给函数的数据不在函数中被改变，就应使用常引用。常引用声明方式：const 类型标识符 &引用名=目标变量名；

```
例1
int a ;
const int &ra=a;
ra=1; //错误
a=1; //正确
例2
string foo( );
void bar(string & s);
那么下面的表达式将是非法的：
bar(foo( ));
bar("hello world");
```

原因在于foo( )和"hello world"串都会产生一个临时对象，而在C++中，这些临时对象都是const类型的。因此上面的表达式就是试图将一个const类型的对象转换为非const类型，这是非法的。引用型参数应该在能被定义为const的情况下，尽量定义为const 。

10、将“引用”作为函数返回值类型的格式、好处和需要遵守的规则?

格式：类型标识符 &函数名（形参列表及类型说明）{ //函数体 }

好处：在内存中不产生被返回值的副本；（注意：正是因为这点原因，所以返回一个局部变量的引用是不可取的。因为随着该局部变量生存期的结束，相应的引用也会失效，产生runtime error!

注意事项：

（1）不能返回局部变量的引用。这条可以参照Effective C++[1]的Item 31。主要原因是局部变量会在函数返回后被销毁，因此被返回的引用就成为了"无所指"的引用，程序会进入未知状态。

（2）不能返回函数内部new分配的内存的引用。这条可以参照Effective C++[1]的Item 31。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，但最好是const。这条原则可以参照Effective C++[1]的Item 30。主要原因是当对象的属性是与某种业务规则（business rule）相关联的时候，其赋值常常与某些其它属性或者对象的状态有关，因此有必要将赋值操作封装在一个业务规则当中。如果其它对象可以获得该属性的非常量引用（或指针），那么对该属性的单纯赋值就会破坏业务规则的完整性。

（4）流操作符重载返回值申明为“引用”的作用：

流操作符<<和>>，这两个操作符常常希望被连续使用，例如：cout << "hello" << endl; 因此这两个操作符的返回值应该是一个仍然支持这两个操作符的流引用。可选的其它方案包括：返回一个流对象和返回一个流对象指针。但是对于返回一个流对象，程序必须重新（拷贝）构造一个新的流对象，也就是说，连续的两个<<操作符实际上是针对不同对象的！这无法让人接受。对于返回一个流指针则不能连续使用<<操作符。因此，返回一个流对象引用是惟一选择。这个唯一选择很关键，它说明了引用的重要性以及无可替代性，也许这就是C++语言中引入引用这个概念的原因吧。

赋值操作符=。这个操作符象流操作符一样，是可以连续使用的，例如：x = j = 10;或者(x=10)=100;赋值操作符的返回值必须是一个左值，以便可以被继续赋值。因此引用成了这个操作符的惟一返回值选择。

```
#include<iostream.h>
int &put(int n);
int vals[10];
int error=-1;
void main()
{
   put(0)=10; //以put(0)函数值作为左值，等价于vals[0]=10;
   put(9)=20; //以put(9)函数值作为左值，等价于vals[9]=20;
```

```
    cout<<vals[0];
    cout<<vals[9];
}
int &put(int n)
{
    if (n>=0 && n<=9 ) return vals[n];
    else { cout<<"subscript error"; return error; }
}
```

（5）在另外的一些操作符中，却千万不能返回引用：+-*/ 四则运算符。它们不能返回引用，Effective C++[1]的Item23详细的讨论了这个问题。主要原因是这四个操作符没有side effect，因此，它们必须构造一个对象作为返回值，可选的方案包括：返回一个对象、返回一个局部变量的引用，返回一个new分配的对象的引用、返回一个静态对象引用。根据前面提到的引用作为返回值的三个规则，第2、3两个方案都被否决了。静态对象的引用又因为((a+b) == (c+d))会永远为true而导致错误。所以可选的只剩下返回一个对象了。

11、结构体与联合有何区别?

(1). 结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员（所有成员共用一块地址空间）, 而结构的所有成员都存在（不同成员的存放地址不同）。

(2). 对于联合的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

12、写出程序结果

```
int  a=4;
int  &f(int  x)
{
    a=a+x;
    return  a;
}
int main(void)
{
    int   t=5;
    cout<<f(t)<<endl;      a = 9
```

```
    f(t)=20;              a = 20
    cout<<f(t)<<endl;     t = 5,a = 20  a = 25
    t=f(t);               a = 30 t = 30
    cout<<f(t)<<endl; }   t = 60
}
```

13、重载（overload)和重写(overried，有的书也叫做“覆盖”）的区别?

常考的题目。从定义上来说：

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）。

重写：是指子类重新定义父类虚函数的方法。

从实现原理上来说：

重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数（至少对于编译器来说是这样的）。如，有两个同名函数：function func(p:integer):integer;和function func(p:string):integer;。那么编译器做过修饰后的函数名称可能是这样的：int_func、str_func。对于这两个函数的调用，在编译器间就已经确定了，是静态的。也就是说，它们的地址在编译期就绑定了（早绑定），因此，重载和多态无关!

重写：和多态真正相关。当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。因此，这样的函数地址是在运行期绑定的（晚绑定）。

14、有哪几种情况只能用intialization list 而不能用assignment?(微软面试)

答案：当类中含有const、reference 成员变量；基类无默认构造函数，有参的构造函数都需要初始化（基类的构造函数都需要初始化表）。

[参考]

15、 C++是不是类型安全的?

答案：不是。两个不同类型的指针之间可以强制转换（用reinterpret cast)。C#是类型安全的。

16、main 函数执行以前，还会执行什么代码?

答案：全局对象的构造函数会在main 函数之前执行。

17、描述内存分配方式以及它们的区别?

1、从静态存储区域分配。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。例如全局变量，static 变量。

2、在栈上创建。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。

3、从堆上分配，亦称动态内存分配。程序在运行的时候用malloc 或new 申请任意多少的内存，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，但问题也最多。

18、分别写出BOOL,int,float,指针类型的变量a 与“零”的比较语句。

答案：

```
BOOL : if ( !a ) or if(a)
int : if ( a == 0)
float : const EXPRESSION EXP = 0.000001
if ( a < EXP && a >-EXP)
pointer : if ( a != NULL) or if(a == NULL)
```

19、请说出const与#define 相比，有何优点?

const作用：定义常量、修饰函数参数、修饰函数返回值三个作用。被Const修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

1、const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误。

2、 有些集成化的调试工具可以对const 常量进行调试，但是不能对宏常量进行调试。

20、简述数组与指针的区别?

数组要么在静态存储区被创建（如全局数组），要么在栈上被创建。指针可以随时指向任意类型的内存块。

1、修改内容上的差别

```
char a[] = “hello”;
a[0] = ‘X’;
char *p = “world”; // 注意p 指向常量字符串
p[0] = ‘X’; // 编译器不能发现该错误，运行时错误
```

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。C++/C 语言没有办法知道指针所指的内存容量，除非在申请内存时记住它。注意当数组作为函数的参数进行传递时，该数组自动退化为同类型的指针。

```
char a[] = "hello world";
char *p = a;
cout<< sizeof(a) << endl; // 12 字节  32位和64位都一样
cout<< sizeof(p) << endl; // 32位机器是4，  64位机器是8
// 计算数组和指针的内存容量
void Func(char a[100])
{
   cout<< sizeof(a) << endl; // 4 字节而不是100 字节
}
```

21、int (*s[10])(int) 表示的是什么?

int (*s[10])(int) 函数指针数组，每个指针指向一个int func(int param)的函数。

22、栈内存与文字常量区

```
char str1[] = "abc";
char str2[] = "abc";

```

```
const char str3[] = "abc";
const char str4[] = "abc";

const char *str5 = "abc";
const char *str6 = "abc";

char *str7 = "abc";
char *str8 = "abc";   // 在vs2017中不通过，字面值不能直接赋给char*类型

cout << ( str1 == str2 ) << endl;   //0  分别指向各自的栈内存
cout << ( str3 == str4 ) << endl;   //0  分别指向各自的栈内存
cout << ( str5 == str6 ) << endl;   //1指向文字常量区地址相同

cout << ( str7 == str8 ) << endl;   //1指向文字常量区地址相同

结果是：0 0 1 1
```

解答：str1,str2,str3,str4是数组变量，它们有各自的内存空间；而str5,str6,str7,str8是指针，它们指向相同的常量区域。

23、将程序跳转到指定内存地址

要对绝对地址0x100000赋值，我们可以用(unsigned int*)0x100000 = 1234;那么要是想让程序跳转到绝对地址是0x100000去执行，应该怎么做?

```
*((void (*)( ))0x100000 ) ( );
首先要将0x100000强制转换成函数指针,即:
(void (*)())0x100000
    然后再调用它:
*((void (*)())0x100000)();
用typedef可以看得更直观些:
typedef void(*)() voidFuncPtr;
*((voidFuncPtr)0x100000)();
```

24、int id[sizeof(unsigned long)];这个对吗？为什么?

答案:正确 这个 sizeof是编译时运算符，编译时就确定了 ,可以看成和机器有关的常量。

25、引用与指针有什么区别?

【参考答案】
1、引用必须被初始化，指针不必。

2、引用初始化以后不能被改变，指针可以改变所指的对象。

3、不存在指向空值的引用，但是存在指向空值的指针。

26、const 与 #define 的比较， const有什么优点?

【参考答案】

1、const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到的错误（边际效应） 。

2、有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

27、复杂声明 下面分别表示什么意思?

```
void * ( * (*fp1)(int))[10];

float (*(* fp2)(int,int,int))(int);

int (* ( * fp3)())[10]();
```

【标准答案】

1、void * (* (*fp1)(int))[10]; *fp1是一个指针，指向一个函数，这个函数的参数为int型，函数的返回值是一个指针，这个指针指向一个数组，这个数组有10个元素，每个元素是一个void*型指针。

2、float ((* fp2)(int,int,int))(int); fp2是一个指针，指向一个函数，这个函数的参数为3个int型，函数的返回值是一个指针，这个指针指向一个函数，这个函数的参数为int型，函数的返回值是float型。

3、int (* (* fp3)())10; fp3是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是一个指针，这个指针指向一个数组，这个数组有10个元素，每个元素是一个指针，指向一个函数，这个函数的参数为空，函数的返回值是int型。

28、内存的分配方式有几种?

【参考答案】

1、从静态存储区域分配。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。例如全局变量。

2、在栈上创建。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。

3、从堆上分配，亦称动态内存分配。程序在运行的时候用malloc或new申请任意多少的内存，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我们决定，使用非常灵活，但问题也最多。

29、基类的析构函数不是虚函数，会带来什么问题?

【参考答案】派生类的析构函数用不上，会造成资源的泄漏。

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

【参考答案】

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程序的各个部分都可以用到；局部变量只能在局部使用；分配在栈区。

操作系统和编译器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 。

牛客网上的题

1、写出完整版的strcpy函数

```
1
2  char * strcpy( char *strDest, const char *strSrc )
3  {
4     assert( (strDest != NULL) && (strSrc != NULL) );
5     char *address = strDest;
6     while( (*strDest++ = * strSrc++) != '\0' );
7     return address;
8  }
```

要点：

1、使用assert断言函数，判断参数是否为NULL；

2、遇'\0'则停止赋值；

3、返回新的字符串的首地址。

2、指出代码错误

```
1  void Test( void )
2  {
3   char *str = (char *) malloc( 100 );
4   strcpy(str, "hello");
5   free( str );
6   ... //省略的其它语句
7  }
```

错误有两个：

使用malloc分配内存后，应判断是否分配成功；

free之后，应置str为NULL，防止变成野指针。

PS：malloc函数

malloc函数是一种分配长度为num_bytes字节的内存块的函数，可以向系统申请分配指定size个字节的内存空间。malloc的全称是memory allocation，中文叫动态内存分配，当无法知道内存具体位置的时候，想要绑定真正的内存空间，就需要用到动态的分配内存。

第一、malloc 函数返回的是 void * 类型。

对于C++，如果你写成：p = malloc (sizeof(int)); 则程序无法通过编译，报错：“不能将 void* 赋值给 int * 类型变量”。

所以必须通过 (int *) 来将强制转换。而对于C，没有这个要求，但为了使C程序更方便的移植到C++中来，建议养成强制转换的习惯。

第二、函数的实参为 sizeof(int)，用于指明一个整型数据需要的大小。

在Linux中可以有这样：malloc(0),这是因为Linux中malloc有一个下限值16Bytes，注意malloc(-1)是禁止的；但是在某些系统中是不允许malloc(0)的。

malloc 只管分配内存，并不能对所得的内存进行初始化，所以得到的一片新内存中，其值将是随机的。

3、分别给出Bool，int， float， 指针变量与“零值”比较的if语句。

```
BOOL型变量：if(!var)

int型变量： if(var==0)

float型变量：

const float EPSINON = 0.00001;

if ((x >= - EPSINON) && (x <= EPSINON)

指针变量：if(var==NULL)
```

4、以下为windows NT下的32位C++程序，请计算sizeof的值

```
void Func ( char str[100] )
{
 sizeof( str ) = ?
}
void *p = malloc( 100 );
sizeof ( p ) = ?
```

sizeof(str) = 4
sizeof (p) = 4
Func (char str[100])函数中数组名作为函数形参时，在函数体内，数组名失去了本身的内涵，仅仅只是一个指针；在失去其内涵的同时，它还失去了其常量特性，可以作自增、自减等操作，可以被修改。

但是数组名在不作形参时，仍然代表整个数组，这时的sizeof应返回数组长度。

sizeof返回的单位是字节。对于结构体，sizeof返回可能会有字节填充。结构体的总大小为结构体最宽基本类型成员大小的整数倍。

5、写一个“标准”宏MIN，这个宏输入两个参数并返回较小的一个。另外，当你写下面的代码时会发生什么事? least = MIN(*p++, b);

答案

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

四个注意点：

宏定义中，左侧为宏名和参数，右侧为宏的实现；

在宏的实现中，所有参数应用括号括起来；

整个宏的实现的外面也要用括号括起来；

最后没有分号。

写下如上代码会导致p自增两次。（作为参数自增一次，计算自增一次）

6、ifndef、extern的用法

条件指示符#ifndef 的最主要目的是防止头文件的重复包含和编译。

extern修饰变量的声明。

如果文件a.c需要引用b.c中变量int v，就可以在a.c中声明extern int v，然后就可以引用变量v。

这里需要注意的是，被引用的变量v的链接属性必须是外链接（external）的，也就是说a.c要引用到v，不只是取决于在a.c中声明extern int v，还取决于变量v本身是能够被引用到的。

7、请说出static和const关键字尽可能多的作用

static:

1. 修饰普通变量，修改变量的存储区域和生命周期，使变量存储在静态区，在main函数运行前就分配了空间，如果有初始值就用初始值初始化它，如果没有初始值系统用默认值初始化它。在每次调用时，其值为上一次调用后改变的值，调用结束后不释放空间。此变量只在声明变量的文件内可见。
2. 修饰普通函数，表明函数的作用范围，仅在定义该函数的文件内才能使用。在多人开发项目时，为了防止与他人命令函数重名，可以将函数定义为static。
3. 修饰成员变量，修饰成员变量使所有的对象只保存一个该变量，而且不需要生成对象就可以访问该成员。
4. 修饰成员函数，修饰成员函数使得不需要生成对象就可以访问该函数，但是在static函数内不能访问非静态成员。

const:

1. 修饰变量，说明该变量不可以被改变；
2. 修饰指针，分为指向常量的指针和指针常量；
3. 常量引用，经常用于形参类型，即避免了拷贝，又避免了函数对值的修改；
4. 修饰成员函数，说明该成员函数内不能修改成员变量。

8、请说一下C/C++ 中指针和引用的区别?

1、指针有自己的一块空间，而引用只是一个别名；

2、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；

3、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；

4、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；

5、可以有const指针，但是没有const引用；

6、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；

7、指针可以有多级指针（**p），而引用只有一级；

8、指针和引用使用++运算符的意义不一样；

9、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。

9、给定三角形ABC和一点P(x,y,z)，判断点P是否在ABC内，给出思路并手写代码

根据面积法，如果P在三角形ABC内，那么三角形ABP的面积+三角形BCP的面积+三角形ACP的面积应该等于三角形ABC的面积。

10、野指针是什么?

野指针就是指向一个已删除的对象或者未申请访问受限内存区域的指针。

11、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?

将可能会被继承的父类的析构函数设置为虚函数，可以保证当我们new一个子类，然后使用基类指针指向该子类对象，释放基类指针时可以释放掉子类的空间，防止内存泄漏。

C++默认的析构函数不是虚函数是因为虚函数需要额外的虚函数表和虚表指针，占用额外的内存。而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，而是只有当需要当作父类时，设置为虚函数。

PS：C++类的六个默认成员函数：

构造函数：一个特殊的成员函数，名字与类名相同，创建类类型对象的时候，由编译器自动调用，在对象的生命周期内只且调用一次，以保证每个数据成员都有一个合适的初始值。 拷贝构造函数：只有单个形参，而且该形参是对本类类型对象的引用（常用const修饰），这样的构造函数称为拷贝构造函数。拷贝构造函数是特殊的构造函数，创建对象时使用已存在的同类对象来进行初始化，由编译器自动调用。 析构函数：与构造函数功能相反，在对象被销毁时，由编译器自动调用，完成类的一些资源清理和收尾工作。 赋值运算符重载：对于类类型的对象我们需要对'='重载，以完成类类型对象之间的赋值。 取址操作符重载：函数返回值为该类型的指针，无参数。 const修饰的取址运算符重载。

12、C++中析构函数的作用?

析构函数与构造函数对应，当对象结束其生命周期，如对象所在的函数已调用完毕时，系统会自动执行析构函数。

析构函数名也应与类名相同，只是在函数名前面加一个位取反符，例如stud( )，以区别于构造函数。它不能带任何参数，也没有返回值（包括void类型）。只能有一个析构函数，不能重载。

如果用户没有编写析构函数，编译系统会自动生成一个缺省的析构函数（即使自定义了析构函数，编译器也总是会为我们合成一个析构函数，并且如果自定义了析构函数，编译器在执行时会先调用自定义的析构函数再调用合成的析构函数），它也不进行任何操作。所以许多简单的类中没有用显式的析构函数。

如果一个类中有指针，且在使用的过程中动态的申请了内存，那么最好显示构造析构函数在销毁类之前，释放掉申请的内存空间，避免内存泄漏。

类析构顺序：1）派生类本身的析构函数；2）对象成员析构函数；3）基类析构函数。

13、map和set有什么区别，分别又是怎么实现的?

map和set都是C++的关联容器，其底层实现都是红黑树（RB-Tree）。由于 map 和set所开放的各种操作接口，RB-tree 也都提供了，所以几乎所有的 map 和set的操作行为，都只是转调 RB-tree 的操作行为。

map和set区别在于：

（1）map中的元素是key-value（关键字—值）对：关键字起到索引的作用，值则表示与索引相关联的数据；Set与之相对就是关键字的简单集合，set中每个元素只包含一个关键字。

（2）set的迭代器是const的，不允许修改元素的值；map允许修改value，但不允许修改key。其原因是因为map和set是根据关键字排序来保证其有序性的，如果允许修改key的话，那么首先需要删除该键，然后调节平衡，再插入修改后的键值，调节平衡，如此一来，严重破坏了map和set的结构，导致iterator失效，不知道应该指向改变前的位置，还是指向改变后的位置。所以STL中将set的迭代器设置成const，不允许修改迭代器的值；而map的迭代器则不允许修改key值，允许修改value值。

（3）map支持下标操作，set不支持下标操作。map可以用key做下标，map的下标运算符[ ]将关键码作为下标去执行查找，如果关键码不存在，则插入一个具有该关键码和mapped_type类型默认值的元素至map中，因此下标运算符[ ]在map应用中需要慎用，const_map不能用，只希望确定某一个关键值是否存在而不希望插入元素时也不应该使用，mapped_type类型没有默认值也不应该使用。如果find能解决需要，尽可能用find。

14、C++中类成员的访问权限有哪些?

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。

private和protected的区别是，子类的对象也可以访问protected，但只有本类的对象可以访问private。~~子类的对象也可以访问private，但只有本类的对象可以访问protected~~。

15、C++中struct和class的区别?

在C++中，可以用struct和class定义类，都可以继承。区别在于：struct的默认继承权限和默认访问权限是public，而class的默认继承权限和默认访问权限是private。

16、一个C++源文件从文本到可执行文件经历的过程?

对于C++源文件，从文本到可执行文件一般需要四个过程：

预处理阶段：对源代码文件中文件包含关系（头文件）、预编译语句（宏定义）进行分析和替换，生成预编译文件。

编译阶段：将经过预处理后的预编译文件转换成特定汇编代码，生成汇编文件

汇编阶段：将编译阶段生成的汇编文件转化成机器码，生成可重定位目标文件

链接阶段：将多个目标文件及所需要的库连接成最终的可执行目标文件

17、include头文件的顺序以及双引号""和尖括号<>的区别?

Include头文件的顺序：对于include的头文件来说，如果在文件a.h中声明一个在文件b.h中定义的变量，而不引用b.h。那么要在a.c文件中引用b.h文件，并且要先引用b.h，后引用a.h,否则汇报变量类型未声明错误。

双引号和尖括号的区别：编译器预处理阶段查找头文件的路径不一样。

对于使用双引号包含的头文件，查找头文件路径的顺序为：

当前头文件目录

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

对于使用尖括号包含的头文件，查找头文件的路径顺序为：

编译器设置的头文件路径（编译器可使用-I显式指定搜索路径）

系统变量CPLUS_INCLUDE_PATH/C_INCLUDE_PATH指定的头文件路径

18、malloc的原理，另外brk系统调用和mmap系统调用的作用分别是什么?

Malloc函数用于动态分配内存。为了减少内存碎片和系统调用的开销，malloc其采用内存池的方式，先申请大块内存作为堆区，然后将堆区分为多个内存块，以块作为内存管理的基本单位。当用户申请内存时，直接从堆区分配一块合适的空闲块。Malloc采用隐式链表结构将堆区分成连续的、大小不一的块，包含已分配块和未分配块；同

时malloc采用显示链表结构来管理所有的空闲块，即使用一个双向链表将空闲块连接起来，每一个空闲块记录了一个连续的、未分配的地址。

当进行内存分配时，Malloc会通过隐式链表遍历所有的空闲块，选择满足要求的块进行分配；当进行内存合并时，malloc采用边界标记法，根据每个块的前后块是否已经分配来决定是否进行块合并。

Malloc在申请内存时，一般会通过brk或者mmap系统调用进行申请。其中当申请内存小于128K时，会使用系统函数brk在堆区中分配；而当申请内存大于128K时，会使用系统函数mmap在映射区分配。

19、C++的内存管理是怎样的?

在C++中，虚拟内存分为代码段、数据段、BSS段、堆区、文件映射区以及栈区六部分。

代码段:包括只读存储区和文本区，其中只读存储区存储字符串常量，文本区存储程序的机器代码。

数据段：存储程序中已初始化的全局变量和静态变量

bss 段：存储未初始化的全局变量和静态变量（局部+全局），以及所有被初始化为0的全局变量和静态变量。

堆区：调用new/malloc函数时在堆区动态分配内存，同时需要调用delete/free来手动释放申请的内存。

映射区:存储动态链接库以及调用mmap函数进行的文件映射

栈区：使用栈空间存储函数的返回地址、参数、局部变量、返回值

20、如何判断内存泄漏?

内存泄漏通常是由于调用了malloc/new等内存申请的操作，但是缺少了对应的free/delete。为了判断内存是否泄露，我们一方面可以使用linux环境下的内存泄漏检查工具Valgrind,另一方面我们在写代码时可以添加内存申请和释放的统计功能，统计当前申请和释放的内存是否一致，以此来判断内存是否泄露。

21、什么时候会发生段错误?

段错误通常发生在访问非法内存地址的时候，具体来说分为以下几种情况：

使用野指针

试图修改字符串常量的内容

22、new和malloc的区别?

1、new分配内存按照数据类型进行分配，malloc分配内存按照指定的大小分配；

2、new返回的是指定对象的指针，而malloc返回的是void*，因此malloc的返回值一般都需要进行类型转化。

3、new不仅分配一段内存，而且会调用构造函数，malloc不会。

4、new分配的内存要用delete销毁，malloc要用free来销毁；delete销毁的时候会调用对象的析构函数，而free则不会。

5、new是一个操作符可以重载，malloc是一个库函数。

6、malloc分配的内存不够的时候，可以用realloc扩容。扩容的原理？new没用这样操作。

7、new如果分配失败了会抛出bad_malloc的异常，而malloc失败了会返回NULL。

8、申请数组时： new[]一次分配所有内存，多次调用构造函数，搭配使用delete[]，delete[]多次调用析构函数，销毁数组中的每个对象。而malloc则只能sizeof(int) * n。

23、A* a = new A; a->i = 10;在内核中的内存分配上发生了什么?

1、A *a：a是一个局部变量，类型为指针，故而操作系统在程序栈区开辟4/8字节的空间（0x000m），分配给指针a。

2、new A：通过new动态的在堆区申请类A大小的空间（0x000n）。

3、a = new A：将指针a的内存区域填入栈中类A申请到的地址的地址。即*（0x000m）=0x000n。

4、a->i：先找到指针a的地址0x000m，通过a的值0x000n和i在类a中偏移offset，得到a->i的地址0x000n + offset，进行*(0x000n + offset) = 10的赋值操作，即内存0x000n + offset的值是10。

24、一个类，里面有static，virtual，之类的，来说一说这个类的内存分布?

1、static修饰符

1）static修饰成员变量

对于非静态数据成员，每个类对象都有自己的拷贝。而静态数据成员被当做是类的成员，无论这个类被定义了多少个，静态数据成员都只有一份拷贝，为该类型的所有对象所共享(包括其派生类)。所以，静态数据成员的值对每个对象都是一样的，它的值可以更新。

因为静态数据成员在全局数据区分配内存，属于本类的所有对象共享，所以它不属于特定的类对象，在没有产生类对象前就可以使用。

2）static修饰成员函数

与普通的成员函数相比，静态成员函数由于不是与任何的对象相联系，因此它不具有this指针。从这个意义上来说，它无法访问属于类对象的非静态数据成员，也无法访问非静态成员函数，只能调用其他的静态成员函数。

Static修饰的成员函数，在代码区分配内存。

2、C++继承和虚函数

C++多态分为静态多态和动态多态。静态多态是通过重载和模板技术实现，在编译的时候确定。动态多态通过虚函数和继承关系来实现，执行动态绑定，在运行的时候确定。

动态多态实现有几个条件：

(1) 虚函数；

(2) 一个基类的指针或引用指向派生类的对象；

基类指针在调用成员函数(虚函数)时，就会去查找该对象的虚函数表。虚函数表的地址在每个对象的首地址。查找该虚函数表中该函数的指针进行调用。

每个对象中保存的只是一个虚函数表的指针，C++内部为每一个类维持一个虚函数表，该类的对象的都指向这同一个虚函数表。

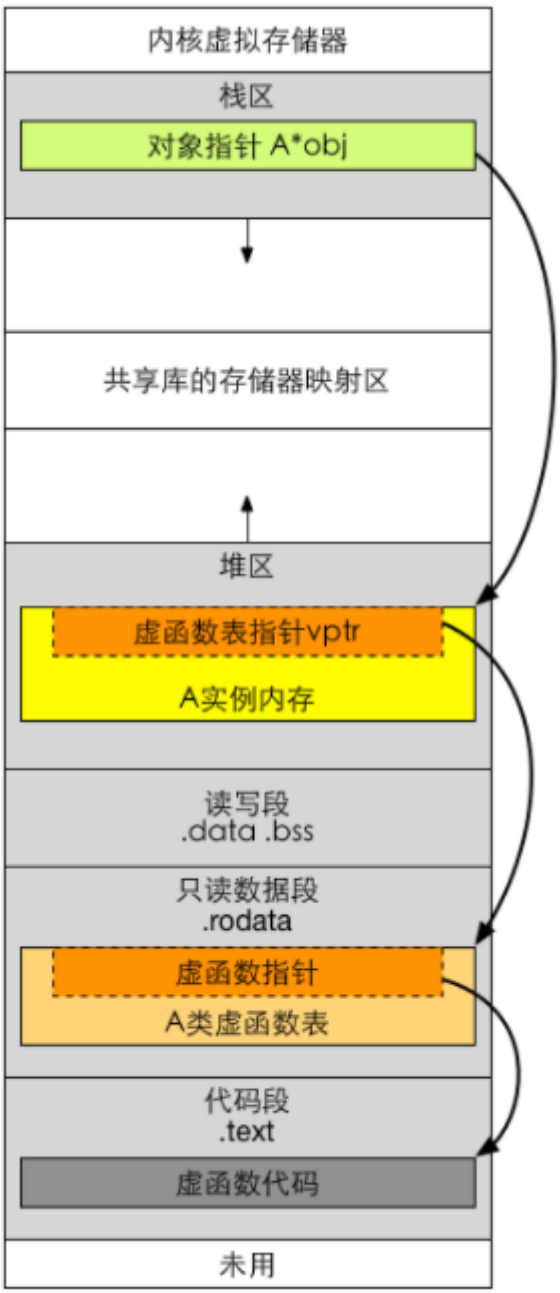
虚函数表中为什么就能准确查找相应的函数指针呢？因为在类设计的时候，虚函数表直接从基类也继承过来，如果覆盖了其中的某个虚函数，那么虚函数表的指针就会被替换，因此可以根据指针准确找到该调用哪个函数。

3、virtual修饰符

如果一个类是局部变量则该类数据存储在栈区，如果一个类是通过new/malloc动态申请的，则该类数据存储在堆区。

如果该类是virutal继承而来的子类，则该类的虚函数表指针和该类其他成员一起存储。虚函数表指针指向只读数据段中的类虚函数表，虚函数表中存放着一个个函数指针，函数指针指向代码段中的具体函数。

如果类中成员是virtual属性，会隐藏父类对应的属性。



25、静态变量什么时候初始化?

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++中由于引入对象，对象生成必须调用构造函数，因此C++规定全局或局部静态对象当且仅当对象首次用到时进行构造。

26、TCP怎么保证可靠性?

TCP保证可靠性：

（1）序列号、确认应答、超时重传

数据到达接收方，接收方需要发出一个确认应答，表示已经收到该数据段，并且确认序号会说明了它下一次需要接收的数据序列号。如果发送发迟迟未收到确认应答，那么可能是发送的数据丢失，也可能是确认应答丢失，这时发送方在等待一定时间后会进行重传。这个时间一般是2*RTT(报文段往返时间）+一个偏差值。

（2）窗口控制与高速重发控制/快速重传（重复确认应答）

TCP会利用窗口控制来提高传输速度，意思是在一个窗口大小内，不用一定要等到应答才能发送下一段数据，窗口大小就是无需等待确认而可以继续发送数据的最大值。如果不使用窗口控制，每一个没收到确认应答的数据都要重发。

使用窗口控制，如果数据段1001-2000丢失，后面数据每次传输，确认应答都会不停地发送序号为1001的应答，表示我要接收1001开始的数据，发送端如果收到3次相同应答，就会立刻进行重发；但还有种情况有可能是数据都收到了，但是有的应答丢失了，这种情况不会进行重发，因为发送端知道，如果是数据段丢失，接收端不会放过它的，会疯狂向它提醒……

（3）拥塞控制

如果把窗口定的很大，发送端连续发送大量的数据，可能会造成网络的拥堵（大家都在用网，你在这狂发，吞吐量就那么大，当然会堵），甚至造成网络的瘫痪。所以TCP在为了防止这种情况而进行了拥塞控制。

慢启动：定义拥塞窗口，一开始将该窗口大小设为1，之后每次收到确认应答（经过一个rtt），将拥塞窗口大小*2。

拥塞避免：设置慢启动阈值，一般开始都设为65536。拥塞避免是指当拥塞窗口大小达到这个阈值，拥塞窗口的值不再指数上升，而是加法增加（每次确认应答/每个rtt，拥塞窗口大小+1），以此来避免拥塞。

将报文段的超时重传看做拥塞，则一旦发生超时重传，我们需要先将阈值设为当前窗口大小的一半，并且将窗口大小设为初值1，然后重新进入慢启动过程。

快速重传：在遇到3次重复确认应答（高速重发控制）时，代表收到了3个报文段，但是这之前的1个段丢失了，便对它进行立即重传。

然后，先将阈值设为当前窗口大小的一半，然后将拥塞窗口大小设为慢启动阈值+3的大小。

这样可以达到：在TCP通信时，网络吞吐量呈现逐渐的上升，并且随着拥堵来降低吞吐量，再进入慢慢上升的过程，网络不会轻易的发生瘫痪。

27、红黑树和AVL树的定义，特点，以及二者区别

平衡二叉树（AVL树）：

平衡二叉树又称为AVL树，是一种特殊的二叉排序树。其左右子树都是平衡二叉树，且左右子树高度之差的绝对值不超过1。一句话表述为：以树中所有结点为根的树的左右子树高度之差的绝对值不超过1。将二叉树上结点的左子树深度减去右子树深度的值称为平衡因子BF，那么平衡二叉树上的所有结点的平衡因子只可能是-1、0和1。只要二叉树上有一个结点的平衡因子的绝对值大于1，则该二叉树就是不平衡的。

红黑树：

红黑树是一种二叉查找树，但在每个节点增加一个存储位表示节点的颜色，可以是红或黑（非红即黑）。通过对任何一条从根到叶子的路径上各个节点着色的方式的限制，红黑树确保没有一条路径会比其它路径长出两倍，因此，红黑树是一种弱平衡二叉树，相对于要求严格的AVL树来说，它的旋转次数少，所以对于搜索，插入，删除操作较多的情况下，通常使用红黑树。

性质：

1. 每个节点非红即黑
2. 根节点是黑的;
3. 每个叶节点（叶节点即树尾端NULL指针或NULL节点）都是黑的;
4. 如果一个节点是红色的，则它的子节点必须是黑色的。
5. 对于任意节点而言，其到叶子点树NULL指针的每条路径都包含相同数目的黑节点;

区别：

AVL 树是高度平衡的，频繁的插入和删除，会引起频繁的rebalance，导致效率下降；红黑树不是高度平衡的，算是一种折中，插入最多两次旋转，删除最多三次旋转。

28、map和unordered_map优点和缺点

对于map，其底层是基于红黑树实现的，优点如下：

1)有序性，这是map结构最大的优点，其元素的有序性在很多应用中都会简化很多的操作

2)map的查找、删除、增加等一系列操作时间复杂度稳定，都为logn

缺点如下：

1）查找、删除、增加等操作平均时间复杂度较慢，与n相关

对于unordered_map来说，其底层是一个哈希表，优点如下：

查找、删除、添加的速度快，时间复杂度为常数级O(c)

缺点如下：

因为unordered_map内部基于哈希表，以（key,value）对的形式存储，因此空间占用率高

Unordered_map的查找、删除、添加的时间复杂度不稳定，平均为O(c)，取决于哈希函数。极端情况下可能为O(n)

29、Top(K)问题

1、直接全部排序（只适用于内存够的情况）

当数据量较小的情况下，内存中可以容纳所有数据。则最简单也是最容易想到的方法是将数据全部排序，然后取排序后的数据中的前K个。

这种方法对数据量比较敏感，当数据量较大的情况下，内存不能完全容纳全部数据，这种方法便不适应了。即使内存能够满足要求，该方法将全部数据都排序了，而题目只要求找出top K个数据，所以该方法并不十分高效，不建议使用。

2、快速排序的变形 （只使用于内存够的情况）

这是一个基于快速排序的变形，因为第一种方法中说到将所有元素都排序并不十分高效，只需要找出前K个最大的就行。

这种方法类似于快速排序，首先选择一个划分元，将比这个划分元大的元素放到它的前面，比划分元小的元素放到它的后面，此时完成了一趟排序。如果此时这个划分元的序号index刚好等于K，那么这个划分元以及它左边的数，刚好就是前K个最大的元素；如果index ＞ K，那么前K大的数据在index的左边，那么就继续递归的从index-1

个数中进行一趟排序；如果index < K，那么再从划分元的右边继续进行排序，直到找到序号index刚好等于K为止。再将前K个数进行排序后，返回Top K个元素。这种方法就避免了对除了Top K个元素以外的数据进行排序所带来的不必要的开销。

3、最小堆法

这是一种局部淘汰法。先读取前K个数，建立一个最小堆。然后将剩余的所有数字依次与最小堆的堆顶进行比较，如果小于或等于堆顶数据，则继续比较下一个；否则，删除堆顶元素，并将新数据插入堆中，重新调整最小堆。当遍历完全部数据后，最小堆中的数据即为最大的K个数。

4、分治法

将全部数据分成N份，前提是每份的数据都可以读到内存中进行处理，找到每份数据中最大的K个数。此时剩下N*K个数据，如果内存不能容纳*NK个数据，则再继续分治处理，分成M份，找出每份数据中最大的K个数，如果M*K个数仍然不能读到内存中，则继续分治处理。直到剩余的数可以读入内存中，那么可以对这些数使用快速排序的变形或者归并排序进行处理。

5、Hash法

如果这些数据中有很多重复的数据，可以先通过hash法，把重复的数去掉。这样如果重复率很高的话，会减少很大的内存用量，从而缩小运算空间。处理后的数据如果能够读入内存，则可以直接排序；否则可以使用分治法或者最小堆法来处理数据。

30、栈和堆的区别，以及为什么栈要快?

堆和栈的区别：

堆是由低地址向高地址扩展；栈是由高地址向低地址扩展

堆中的内存需要手动申请和手动释放；栈中内存是由OS自动申请和自动释放，存放着参数、局部变量等内存

堆中频繁调用malloc和free,会产生内存碎片，降低程序效率；而栈由于其先进后出的特性，不会产生内存碎片

堆的分配效率较低，而栈的分配效率较高

栈的效率高的原因：

栈是操作系统提供的数据结构，计算机底层对栈提供了一系列支持：分配专门的寄存器存储栈的地址，压栈和入栈有专门的指令执行；而堆是由C/C++函数库提供的，机制复杂，需要一系列分配内存、合并内存和释放内存的算法，因此效率较低。

31、写个函数在main函数执行前先运行

```
__attribute((constructor))void before()
{
    printf("before main\n");
}
```

32、extern“C”的作用?

C++调用C函数需要extern C，因为C语言没有函数重载。

33、STL迭代器删除元素

1、对于序列容器vector,deque来说，使用erase(itertor)后，后边的每个元素的迭代器都会失效，但是后边每个元素都会往前移动一个位置，但是erase会返回下一个有效的迭代器；

2、对于关联容器map set来说，使用了erase(iterator)后，当前元素的迭代器失效，但是其结构是红黑树，删除当前元素的，不会影响到下一个元素的迭代器，所以在调用erase之前，记录下一个元素的迭代器即可。

3、对于list来说，它使用了不连续分配的内存，并且它的erase方法也会返回下一个有效的iterator。

34、vector和list的区别与应用有哪些?

1、概念：

1）Vector

连续存储的容器，动态数组，在堆上分配空间

底层实现：数组

两倍容量增长：

vector 增加（插入）新元素时，如果未超过当时的容量，则还有剩余空间，那么直接添加到最后（插入指定位置），然后调整迭代器。

如果没有剩余空间了，则会重新配置原有元素个数的两倍空间，然后将原空间元素通过复制的方式初始化新空间，再向新空间增加元素，最后析构并释放原空间，之前的迭代器会失效。

性能：

访问：O(1)

插入：在最后插入（空间够）：很快

在最后插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

在中间插入（空间够）：内存拷贝

在中间插入（空间不够）：需要内存申请和释放，以及对之前数据进行拷贝。

删除：在最后删除：很快

在中间删除：内存拷贝

适用场景：经常随机访问，且不经常对非尾节点进行插入删除。

2、List

动态链表，在堆上分配空间，每插入一个元数都会分配空间，每删除一个元素都会释放空间。

底层：双向链表

性能：

访问：随机访问性能很差，只能快速访问头尾节点。

插入：很快，一般是常数开销

删除：很快，一般是常数开销

适用场景：经常插入删除大量数据

2、区别：

1）vector底层实现是数组；list是双向 链表。

2）vector支持随机访问，list不支持。

3）vector是顺序内存，list不是。

4）vector在中间节点进行插入删除会导致内存拷贝，list不会。

5）vector一次性分配好内存，不够时才进行2倍扩容；list每次插入新节点都会进行内存申请。

6）vector随机访问性能好，插入删除性能差；list随机访问性能差，插入删除性能好。

3、应用

vector拥有一段连续的内存空间，因此支持随机访问，如果需要高效的随即访问，而不在乎插入和删除的效率，使用vector。

list拥有一段不连续的内存空间，如果需要高效的插入和删除，而不关心随机访问，则应使用list。

35、STL里resize和reserve的区别?

resize()：改变当前容器内含有元素的数量(size())，eg: vectorv; v.resize(len);v的size变为len,如果原来v的size小于len，那么容器新增（len-size）个元素，元素的值为默认为0.当v.push_back(3);之后，则是3是放在了v的末尾，即下标为len，此时容器是size为len+1； reserve()：改变当前容器的最大容量（capacity）,它不会生成元素，只是确定这个容器允许放入多少对象，如果reserve(len)的值大于当前的capacity()，那么会重新分配一块能存len个对象的空间，然后把之前v.size()个对象通过copy construtor复制过来，销毁之前的内存。

36、源码到可执行文件的过程?

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：

1、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。

2、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。

37、tcp握手为什么两次不可以？为什么不用四次?

两次不可以：tcp是全双工通信，两次握手只能确定单向数据链路是可以通信的，并不能保证反向的通信正常

不用四次： 本来握手应该和挥手一样都是需要确认两个方向都能联通的，本来模型应该是： 1.客户端发送syn0给服务器 2.服务器收到syn0，回复ack(syn0+1) 3.服务器发送syn1 4.客户端收到syn1，回复ack(syn1+1) 因为tcp是全双工的，上边的四部确认了数据在两个方向上都是可以正确到达的，但是2，3步没有没有上下的联系，可以将其合并，加快握手效率，所有就变成了3步握手。

面试常见题

1、什么是虚函数？什么是纯虚函数?

虚函数是允许被其子类重新定义的成员函数。

虚函数的声明：virtual returntype func(parameter);引入虚函数的目的是为了动态绑定;

纯虚函数声明：virtual returntype func(parameter)=0；引入纯虚函数是为了派生接口。（使派生类仅仅只是继承函数的接口）

2、基类为什么需要虚析构函数?

防止内存泄漏。想去借助父类指针去销毁子类对象的时候，不能去销毁子类对象。假如没有虚析构函数，释放一个由基类指针指向的派生类对象时，不会触发动态绑定，则只会调用基类的析构函数，不会调用派生类的。派生类中申请的空间则得不到释放导致内存泄漏。

3、当i是一个整数的时候i++和++i那个更快？它们的区别是什么?

几乎一样。i++返回的是i的值，++i返回的是i+1的值，即++i是一个确定的值，是一个可以修改的左值。

4、vector的reserve和capacity的区别?

reserve()用于让容器预留空间，避免再次分配内存；capacity()返回在重新进行分配以前所能容纳的元素数量。

5、如何初始化const和static数据成员?

通常在类外申明static成员，但是static const的整型（bool，char，int，long）可以在类中声明且初始化，static const的其他类型必须在类外初始化（包括整型数组）。

6、static 和const分别怎么用，类里面static和const可以同时修饰成员函数吗?

static的作用：

对变量：

1.局部变量：

在局部变量之前加上关键字static，局部变量就被定义成为一个局部静态变量。

1）内存中的位置：静态存储区

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域随之结束。

注：当static用来修饰局部变量的时候，它就**改变了局部变量的存储位置（从原来的栈中存放改为静态存储区）及其生命周期（局部静态变量在离开作用域之后，并没有被销毁，而是仍然驻留在内存当中，直到程序结束，只不过我们不能再对他进行访问），但未改变其作用域。**

2.全局变量

在全局变量之前加上关键字static，全局变量就被定义成为一个全局静态变量。

1）内存中的位置：静态存储区（静态存储区在整个程序运行期间都存在）

2）初始化：未经初始化的全局静态变量会被程序自动初始化为0（自动对象的值是任意的，除非他被显示初始化）

3）作用域：全局静态变量在声明他的文件之外是不可见的。准确地讲从定义之处开始到文件结尾。

注：static修饰全局变量，**并未改变其存储位置及生命周期，而是改变了其作用域，使当前文件外的源文件无法访问该变量**，好处如下：（1）不会被其他文件所访问，修改（2）其他文件中可以使用相同名字的变量，不会发生冲突。**对全局函数也是有隐藏作用。**而普通全局变量只要定义了，任何地方都能使用，使用前需要声明所有的.c文件，只能定义一次普通全局变量，但是可以声明多次（外部链接）。注意：全局变量的作用域是全局范围，但是在某个文件中使用时，必须先声明。

对类中的：

1.成员变量

用static修饰类的数据成员实际使其成为类的全局变量，会被类的所有对象共享，包括派生类的对象。因此，**static成员必须在类外进行初始化(初始化格式： int base::var=10;)，而不能在构造函数内进行初始化，不过也可以用const修饰static数据成员在类内初始化 。因为静态成员属于整个类，而不属于某个对象，如果在类内初始化，会导致每个对象都包含该静态成员，这是矛盾的。**

特点：

1.不要试图在头文件中定义(初始化)静态数据成员。在大多数的情况下，这样做会引起重复定义这样的错误。即使加上#ifndef #define #endif或者#pragma once也不行。

2.静态数据成员可以成为成员函数的可选参数，而普通数据成员则不可以。

3.静态数据成员的类型可以是所属类的类型，而普通数据成员则不可以。普通数据成员的只能声明为 所属类类型的指针或引用。

2.成员函数

1. 用static修饰成员函数，使这个类只存在这一份函数，所有对象共享该函数，不含this指针。
2. 静态成员是可以独立访问的，也就是说，无须创建任何对象实例就可以访问。base::func(5,3);当static成员函数在类外定义时不需要加static修饰符。
3. 在静态成员函数的实现中不能直接引用类中说明的非静态成员，可以引用类中说明的静态成员。因为静态成员函数不含this指针。

不可以同时用const和static修饰成员函数。

C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。

我们也可以这样理解：两者的语意是矛盾的。**static的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态**，与类型的静态变量没有关系。因此不能同时用它们。

const的作用：

1.限定变量为不可修改。

2.限定成员函数不可以修改任何数据成员。

3.const与指针：

const char *p 表示 指向的内容不能改变。

char * const p，就是将P声明为常指针，它的地址不能改变，是固定的，但是它的内容可以改变。

7、指针和引用的区别

本质上的区别是，指针是一个新的变量，只是这个变量存储的是另一个变量的地址，我们通过访问这个地址来修改变量。

而引用只是一个别名，还是变量本身。对引用进行的任何操作就是对变量本身进行操作，因此以达到修改变量的目的。

注：

(1)指针：指针是一个变量，只不过这个变量存储的是一个地址，指向内存的一个存储单元；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已。如： int a=1;int *p=&a; int a=1;int &b=a; 上面定义了一个整形变量和一个指针变量p，该指针变量指向a的存储单元，即p的值是a存储单元的地址。

而下面2句定义了一个整形变量a和这个整形a的引用b，事实上a和b是同一个东西，在内存占有同一个存储单元。

(2)可以有const指针，但是没有const引用（const引用可读不可改，与绑定对象是否为const无关）

注：引用可以指向常量，也可以指向变量。例如int &a=b，使引用a指向变量b。而为了让引用指向常量，必须使用常量引用，如const int &a=1; 它代表的是引用a指向一个const int型，这个int型的值不能被改变，而不是引用a的指向不能被改变，因为引用的指向本来就是不可变的，无需加const声明。即指针存在常量指针int const *p和指针常量int *const p，而引用只存在常量引用int const &a，不存在引用常量int& const a。

(3)指针可以有多级，但是引用只能是一级（int **p；合法 而 int &&a是不合法的）

(4)指针的值可以为空，但是引用的值不能为NULL，并且引用在定义的时候必须初始化；

(5)指针的值在初始化后可以改变，即指向其它的存储单元，而引用在进行初始化后就不会再改变了。

(6)"sizeof引用"得到的是所指向的变量(对象)的大小，而"sizeof指针"得到的是指针本身的大小；

(7)指针和引用的自增(++)运算意义不一样；

(8)指针使用时需要解引用（*），引用则不需要；

8、什么是多态？多态有什么用途？（有重载、重写、隐藏的区别）

C++ 多态有两种：静态多态（早绑定）、动态多态（晚绑定）。静态多态是通过函数重载实现的；动态多态是通过虚函数实现的。

1.定义：“一个接口，多种方法”，程序在运行时才决定要调用的函数。

2.实现：C++多态性主要是通过虚函数实现的，虚函数允许子类重写override(注意和overload的区别，overload是重载，是允许同名函数的表现，这些函数参数列表/类型不同）。

注：多态与非多态的实质区别就是函数地址是静态绑定还是动态绑定。如果函数的调用在编译器编译期间就可以确定函数的调用地址，并产生代码，说明地址是静态绑定的；如果函数调用的地址是 需要在运行期间才确定，属于动态绑定。

3.目的：**接口重用。**封装可以使得代码模块化，继承可以扩展已存在的代码，他们的目的都是为了代码重用。而多态的目的则是为了接口重用。

4.用法：声明基类的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而实现不同的方法。

用一句话概括：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。

关于重载、重写、隐藏的区别

Overload(重载)：在C++程序中，可以将语义、功能相似的几个函数用同一个名字表示，但参数或返回值不同（包括类型、顺序不同），即函数重载。 （1）相同的范围（在同一个类中）； （2）函数名字相同； （3）参数不同； （4）virtual 关键字可有可无。

Override(覆盖或重写)：是指派生类函数覆盖基类函数，特征是： （1）不同的范围（分别位于派生类与基类）； （2）函数名字相同； （3）参数相同； （4）基类函数必须有virtual 关键字。 注：重写基类虚函数的时候，会自动转换这个函数为virtual函数，不管有没有加virtual，因此重写的时候不加virtual也是可以的，不过为了易读性，还是加上比较好。

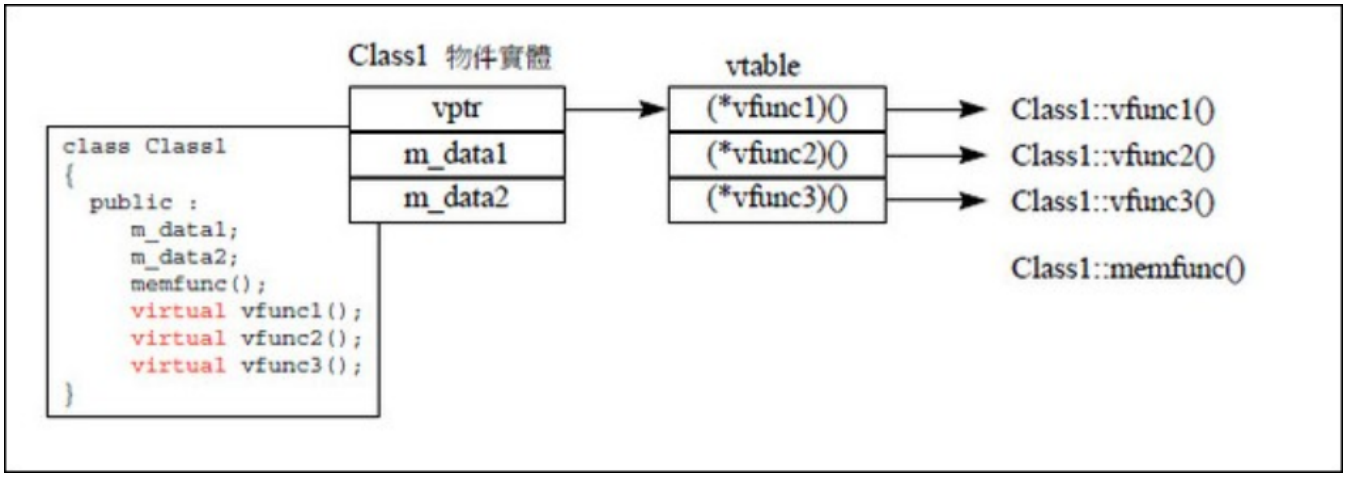
Overwrite(重写)：隐藏，是指派生类的函数屏蔽了与其同名的基类函数，规则如下： （1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。 （2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

虚函数表：

详细解释参考可其他博客

多态是由虚函数实现的，而虚函数主要是通过**虚函数表（V-Table）**来实现的。

如果一个类中包含虚函数（virtual修饰的函数），那么这个类就会包含一张虚函数表，虚函数表存储的每一项是一个虚函数的地址。如下图：

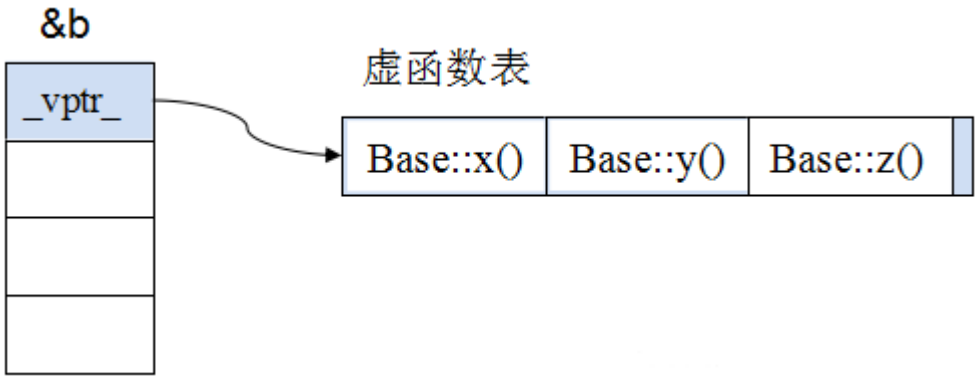


这个类的每一个对象都会包含一个**虚指针**（虚指针存在于对象实例地址的最前面，保证虚函数表有最高的性能），这个虚指针指向虚函数表。

注：对象不包含虚函数表，只有虚指针，类才包含虚函数表，派生类会生成一个兼容基类的虚函数表。

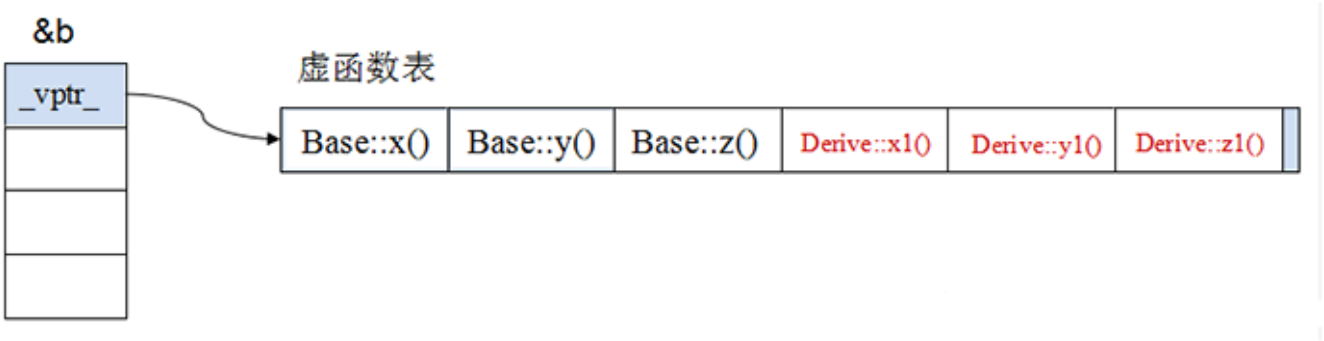
- 原始基类的虚函数表

下图是原始基类的对象，可以看到虚指针在地址的最前面，指向基类的虚函数表（假设基类定义了3个虚函数）



- 单继承时的虚函数（无重写基类虚函数）

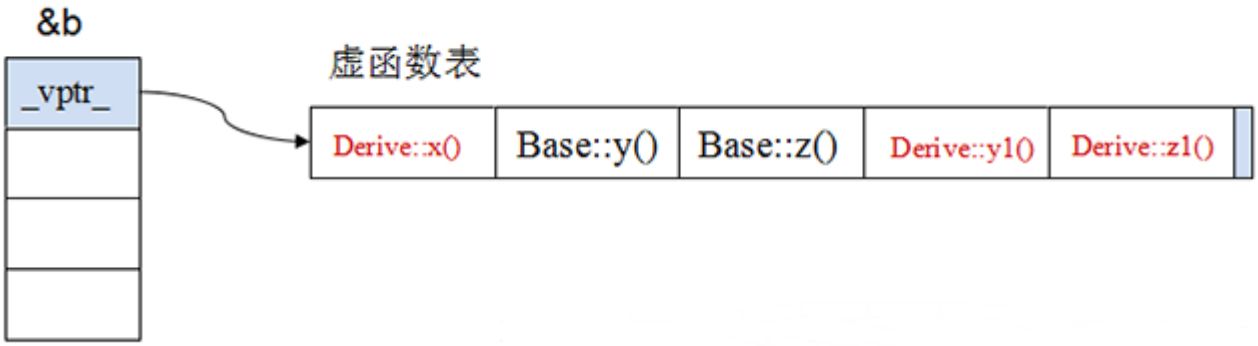
假设现在派生类继承基类，并且重新定义了3个虚函数，派生类会自己产生一个兼容基类虚函数表的**属于自己的虚函数表**。



Derive Class继承了Base Class中的3个虚函数，准确说是该函数的实体地址被拷贝到Derive Class的虚函数列表中，派生新增的虚函数置于虚函数列表后面，并按声明顺序摆放。

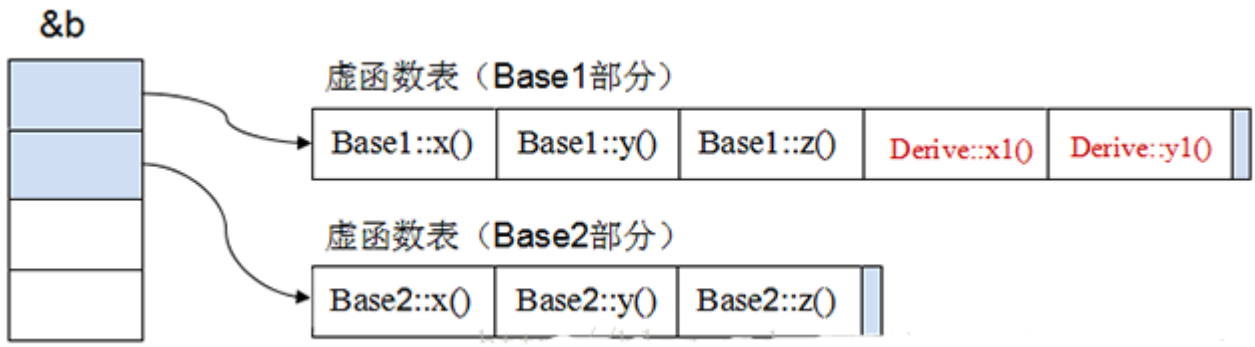
- 单继承时的虚函数（重写基类虚函数）

现在派生类重写基类的x函数，可以看到这个派生类构建自己的虚函数表的时候，修改了base::x()这一项，指向了自己的虚函数。



- 多重继承时的虚函数（class Derived :public Base1,public Base2）

这个派生类多重继承了两个基类base1，base2，因此它有两个虚函数表。



它的对象会有**多个虚指针（据说和编译器相关）**，指向不同的虚函数表。

注：有关以上虚函数表等详见c++对象模型。链接地址：https://www.cnblogs.com/inception6-lxc/p/9273918.html

纯虚函数：

定义： 在很多情况下，基类本身生成对象是不合情理的。为了解决这个问题，方便使用类的多态性，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**;）纯虚函数不能再在基类中实现，编译器要求在派生类中必须予以重写以实现多态性。同时含有纯虚拟函数的类称为抽象类，它不能生成对象。称带有纯虚函数的类为抽象类。

特点：

1，当想在基类中抽象出一个方法，且该基类只做能被继承，而不能被实例化；（避免类被实例化且在编译时候被发现，可以采用此方法）

2，这个方法必须在派生类(derived class)中被实现；

目的：使派生类仅仅只是继承函数的接口。

9、vector中size()和capacity()的区别。

size()指容器当前拥有的元素个数（对应的resize(size_type)会在容器尾添加或删除一些元素，来调整容器中实际的内容，使容器达到指定的大小。）；capacity（）指容器在必须分配存储空间之前可以存储的元素总数。

size表示的这个vector里容纳了多少个元素，capacity表示vector能够容纳多少元素，它们的不同是在于vector的size是2倍增长的。如果vector的大小不够了，比如现在的capacity是4，插入到第五个元素的时候，发现不够了，此时会给他重新分配8个空间，把原来的数据及新的数据复制到这个新分配的空间里。（会有迭代器失效的问题）

10、new和malloc的区别。

本文章中有一个题了。

- new是**运算符**，malloc()是一个**库函数**；
- new会调用**构造函数**，malloc不会；
- new返回指定类型指针，malloc返回**void***指针，需要强制类型转换；
- new会自动计算需分配的空间，malloc不行；
- new可以被**重载**，malloc不能。

11、C++的内存分区

win中，linux中比这个要更加细节，具体有另一篇文章

- **栈区（stack）**：主要存放函数参数以及局部变量，由系统自动分配释放。
- **堆区（heap）**：由用户通过 malloc/new 手动申请，手动释放。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- **全局/静态区**：存放全局变量、静态变量；程序结束后由系统释放。
- **字符串常量区**：字符串常量就放在这里，程序结束后由系统释放。
- **代码区**：存放程序的二进制代码。

12、vector、map、multimap、unordered_map、unordered_multimap的底层数据结构，以及几种map容器如何选择?

底层数据结构：

- vector基于**数组**，map、multimap基于**红黑树**，unordered_map、unordered_multimap基于**哈希表**。

根据应用场景进行选择：

- map/unordered_map **不允许重复元素**
- multimap/unordered_multimap **允许重复元素**
- map/multimap **底层基于红黑树，元素自动有序，且插入、删除效率高**
- unordered_map/unordered_multimap **底层基于哈希表，故元素无序，查找效率高。**

13、内存泄漏怎么产生的？如何避免？

- 内存泄漏一般是指**堆内存**的泄漏，也就是程序在运行过程中动态申请的内存空间不再使用后没有及时释放，导致那块内存不能被再次使用。
- 更广义的内存泄漏还包括未对**系统资源**的及时释放，比如句柄、socket等没有使用相应的函数释放掉，导致系统资源的浪费。

VS中检测内存泄露的方法：

其一： 直接用vs看内存是否暴增不降

其二：

```
#define CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
//在入口函数中包含 _CrtDumpMemoryLeaks();
//即可检测到内存泄露

//以如下测试函数为例：
int main()
{
    char* pChars = new char[10];
    //delete[]pChars;
    _CrtDumpMemoryLeaks();
    system("pause");
    return 0;
}
```

解决方法：

- 养成良好的编码习惯和规范，记得及时释放掉内存或系统资源。
- 重载new和delete，以链表的形式自动管理分配的内存。

- 使用**智能指针**，share_ptr、auto_ptr、weak_ptr。

14、说几个C++11的新特性

- auto类型推导
- 范围for循环
- lambda函数
- override 和 final 关键字

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

class A
{
   virtual void foo();
};
class B :A
{
   virtual void f00(); //OK
   virtual void f0o()override; //Error
};
```

```
/*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：*/

```

```
class Base
{
    virtual void foo();
};

class A : 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
    void bar() final; // Error: 父类中没有 bar虚函数可以被重写或final
};

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
{
};
```

- 空指针常量nullptr
- 线程支持、智能指针等

15、C和C++区别?

C++在C的基础上增添类，C是一个结构化语言，它的重点在于算法和数据结构。C程序的设计首要考虑的是如何通过一个过程，对输入（或环境条件）进行运算处理得到输出（或实现过程（事务）控制），而对于C++，首要考虑的是如何构造一个对象模型，让这个模型能够契合与之对应的问题域，这样就可以通过获取对象的状态信息得到输出或实现过程（事务）控制。

16、const与#define的区别 (文章上面也有)

1.编译器处理方式 define – 在预处理阶段进行替换 const – 在编译时确定其值

2.类型检查 define – 无类型，不进行类型安全检查，可能会产生意想不到的错误 const – 有数据类型，编译时会进行类型检查

3.内存空间 define – 不分配内存，给出的是立即数，有多少次使用就进行多少次替换，在内存中会有多个拷贝，消耗内存大 const – 在静态存储区中分配空间，在程序运行过程中内存中只有一个拷贝

4.其他 在编译时， 编译器通常不为const常量分配存储空间，而是将它们保存在符号表中，这使得它成为一个编译期间的常量，没有了存储与读内存的操作，使得它的效率也很高。 宏替换只作替换，不做计算，不做表达式求解。

17、悬空指针与野指针区别

- 悬空指针：当所指向的对象被释放或者收回，但是没有让指针指向NULL；

```
{
  char *dp = NULL;
  {
     char c;
     dp = &c;
  }
 //变量c释放，dp变成空悬指针
}
```

```
void func()
{
   char *dp = (char *)malloc(A_CONST);
   free(dp);       //dp变成一个空悬指针
   dp = NULL;      //dp不再是空悬指针
   /* ... */
}
```

- 野指针：那些未初始化的指针；

```
int func()
{
   char *dp;//野指针，没有初始化
   static char *sdp;//非野指针，因为静态变量会默认初始化为0
}
```

18、struct与class的区别?

本质区别是**访问的默认控制**：默认的继承访问权限，class是private，struct是public；

19、sizeof和strlen的区别?

功能不同：

sizeof是操作符，参数为任意类型，主要计算类型占用内存大小。

strlen（）是函数，其函数原型为：extern unsigned int strlen(char *s）;其参数为char*,strlen只能计算以"\0"结尾字符串的长度，计算结果不包括"\0"。

```
char* ss="0123456789";
//s1=4,ss为字符指针在内存中占用4个字节
int s1=sizeof(ss);
//s2=10,计算字符串ss的长度
int s2=strlen(ss);
```

参数不同：

当将字符数组作为sizeof（）的参数时，计算字符数组占用内存大小；当将字符数组作为strlen（)函数，字符数组转化为char*。因为sizeof的参数为任意类型，而strlen（）函数参数只能为char*，当参数不是char*必须转换为char*。

```
char str[]="abced";
//a为6（1*6），字符数组str包含6个元素（a,b,c,d,e,\0），每个元素占用1个字节
int a= sizeof(str);
//len为5，不包含"\0",
int len=strlen(str);
//str[0]是字符元素a，所以b=1
int b= sizeof(str[0]);
```

20、32位，64位系统中，各种常用内置数据类型占用的字节数?

char ： 1个字节(固定)

(即指针变量): 4个字节(32位机的寻址空间是4个字节。同理64位编译器)(变化)

short int : 2个字节(固定)

int: 4个字节(固定)

unsigned int : 4个字节(固定)

float: 4个字节(固定)

double: 8个字节(固定)

long: 4个字节

unsigned long: 4个字节(变化*,其实就是寻址控件的地址长度数值)

long long: 8个字节(固定)

64位操作系统

char ： 1个字节(固定)

*(即指针变量): 8个字节

short int : 2个字节(固定)

int: 4个字节(固定)

unsigned int : 4个字节(固定)

float: 4个字节(固定)

double: 8个字节(固定)

long: 8个字节

unsigned long: 8个字节(变化*其实就是寻址控件的地址长度数值)

long long: 8个字节(固定)

除*与long 不同其余均相同。

21、virtual, inline, decltype,volatile,static, const关键字的作用？使用场景？

inline：在c/c++中，**为了解决**一些频繁调用的小函数大量消耗栈空间（栈内存）的问题，特别的引入了inline修饰符，表示为内联函数。

```
#include <stdio.h>
//函数定义为inline即:内联函数
inline char* dbtest(int a) {
   return (i % 2 > 0) ? "奇" : "偶";
}

int main()
{
  int i = 0;
  for (i=1; i < 100; i++) {
     printf("i:%d   奇偶性:%s /n", i, dbtest(i));
  }
}//在for循环的每个dbtest(i)的地方替换成了 (i % 2 > 0) ? "奇" : "偶"，避免了频繁调用函数，对栈内存的消耗
```

decltype: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的值类型。

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。(在多线程中有获取最新值的用法)

static:

1. 隐藏

   在变量和函数名前面如果未加static，则它们是全局可见的。加了static，就会对其它源文件隐藏，利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲 突。static可以用作函数和变量的前缀，对于函数来讲，static的作用仅限于隐藏 。

   2.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。

PS：如果作为static局部变量在函数内定义，它的生存期为整个源程序，但是其作用域仍与自动变量相同，只能在定义该变量的函数内使用该变量。退出该函数后， 尽管该变量还继续存在，但不能使用它。

```
#include <stdio.h>

int fun(){
   static int count = 10;     //在第一次进入这个函数的时候，变量a被初始化为10！并接着自减1，以后每次进入该函数，a
   return count--;            //就不会被再次初始化了，仅进行自减1的操作；在static发明前，要达到同样的功能，则只能使用全局变量：

}

int count = 1;

int main(void)
{
    printf("global\t\tlocal static\n");
    for(; count <= 10; ++count)
            printf("%d\t\t%d\n", count, fun());
    return 0;
}
```

---基于以上两点可以得出一个结论：把局部变量改变为静态变量后是改变了它的存储方式即**改变了它的生存期。**把全局变量改变为静态变量后是改变了它的作用域， **限制了它的使用范围。**因此static 这个说明符在不同的地方所起的作用是不同的。

3.static的第三个作用是默认初始化为0（static变量）

最后对static的三条作用做一句话总结。首先static的最主要功能是隐藏，其次因为static变量存放在静态存储区，所以它具备持久性和默认值0。

4.static的第四个作用：C++中的类成员声明static（有些地方与以上作用重叠）

在类中声明static变量或者函数时，初始化时使用作用域运算符来标明它所属类，因此，静态数据成员是类的成员，而不是对象的成员，这样就出现以下作用：

(1)类的静态成员函数是属于整个类而非类的对象，所以它没有this指针，这就导致 了它仅能访问类的静态数据和静态成员函数。

(2)不能将静态成员函数定义为虚函数。

(3)由于静态成员声明于类中，操作于其外，所以对其取地址操作，就多少有些特殊，变量地址是指向其数据类型的指针 ，函数地址类型是一个“nonmember函数指针”。

(4)由于静态成员函数没有this指针，所以就差不多等同于nonmember函数，结果就产生了一个意想不到的好处：成为一个callback函数，使得我们得以将C++和C-based X W indow系统结合，同时也成功的应用于线程函数身上。 （这条没遇见过)

(5)static并没有增加程序的时空开销，相反她还缩短了子类对父类静态成员的访问 时间，节省了子类的内存空间。

(6)静态数据成员在<定义或说明>时前面加关键字static。

(7)静态数据成员是静态存储的，所以必须对它进行初始化。 （程序员手动初始化，否则编译时一般不会报错，但是在Link时会报错误）

(8)静态成员初始化与一般数据成员初始化不同:

初始化在类体外进行，而前面不加static，以免与一般静态变量或对象相混淆； 初始化时不加该成员的访问权限控制符private，public等；
初始化时使用作用域运算符来标明它所属类； 所以我们得出静态数据成员初始化的格式： **<数据类型><类名>::<静态数据成员名>=<值>**

(9)为了防止父类的影响，可以在子类定义一个与父类相同的静态变量，以屏蔽父类的影响。这里有一点需要注意：**我们说静态成员为父类和子类共享，但我们有重复定义了静态成员，这会不会引起错误呢？不会，我们的编译器采用了一种绝妙的手法：name-mangling 用以生成唯一的标志。**

22、深拷贝与浅拷贝的区别?

1、什么时候用到拷贝函数?

a.一个对象以值传递的方式传入函数体； b.一个对象以值传递的方式从函数返回；

c.一个对象需要通过另外一个对象进行初始化。

如果在类中没有显式地声明一个拷贝构造函数，那么，编译器将会自动生成一个默认的拷贝构造函数，该构造函数完成对象之间的位拷贝。位拷贝又称浅拷贝；

2、是否应该自定义拷贝函数?

自定义拷贝构造函数是一种良好的编程风格，它可以阻止编译器形成默认的拷贝构造函数，提高源码效率。

3、什么叫深拷贝？什么是浅拷贝？两者异同?

如果一个类拥有资源，当这个类的对象发生复制过程的时候，资源重新分配，这个过程就是深拷贝，反之，没有重新分配资源，就是浅拷贝。

4、深拷贝好还是浅拷贝好?

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。

23、派生类中构造函数，析构函数调用顺序?

构造函数：“先基后派”；析构函数：“先派后基”。

24、C++类中数据成员初始化顺序?

1、成员变量在使用初始化列表初始化时，与构造函数中初始化成员列表的顺序无关，只与定义成员变量的顺序有关。

2、如果不使用初始化列表初始化，在构造函数内初始化时，此时与成员变量在构造函数中的位置有关。

3、类中const成员常量必须在构造函数初始化列表中初始化。

4、类中static成员变量，只能在类内外初始化(同一类的所有实例共享静态成员变量)。

初始化顺序：

- 1） 基类的静态变量或全局变量
- 2） 派生类的静态变量或全局变量
- 3） 基类的成员变量
- 4） 派生类的成员变量

25、结构体内存对齐问题？结构体/类大小的计算？

注：内存对齐是看类型，而不是看总的字节数。比如：

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

struct AlignData1
{
    int a;
    char b[7];//a后面并不会补上3个字节，而是由于char的类型所以不用补。
    short c;
    char d;
}Node;
struct AlignData2
{
    bool a;
    int b[2];//a后面并不会补上7个字节，而是根据int的类型补3个字节。
    int c;
    int d;
}Node2;
int main(){
    cout << sizeof(Node) << endl;//16
    cout << sizeof(Node2) << endl;//20
    system("pause");
    return 0;
}
```

补充：

- **每个成员相对于这个结构体变量地址的偏移量正好是该成员类型所占字节的整数倍。**为了对齐数据，可能必须在上一个数据结束和下一个数据开始的地方插入一

些没有用处字节。
- 最终占用字节数为成员类型中**最大占用字节数的整数倍。**
- 一般的结构体成员按照默认对齐字节数递增或是递减的顺序排放，会使总的填充字节数最少。

```
struct AlignData1
{
    char c;
    short b;
    int i;
    char d;
}Node;

这个结构体在编译以后，为了字节对齐，会被整理成这个样子：
struct AlignData1
{
    char c;
    char padding[1];
    short b;
    int i;
    char d;
    char padding[3];
}Node;
```

含虚函数的类的大小：可看其他博客

联合体的大小计算：

联合体所占的空间不仅取决于最宽成员，还跟所有成员有关系，即其大小必须满足两个条件：

1)大小足够容纳最宽的成员；

2)大小能被其包含的所有基本数据类型的大小所整除。

```
union U1
{
   int n;
   char s[11];
   double d;
}; //16，char s[11]按照char=1可以整除

union U2
{
   int n;
   char s[5];
   double d;
}; //8
```

26、static_cast, dynamic_cast, const_cast, reinpreter_cast的区别?

补充：static_cast与dynamic_cast

- cast发生的时间不同，一个是static**编译时**，一个是runtime**运行时**；
- static_cast是相当于C的**强制类型转换**，用起来可能有一点危险，不提供运行时的检查来确保转换的安全性。
- dynamic_cast用于**转换指针和和引用**，**不能用来转换对象** ——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。在多态类型之间的转换主要使用dynamic_cast，因为类型**提供了运行时信息。**

```
#include <iostream>
using namespace std;
class CBasic
{
public:
    virtual int test(){return 0;}
};

```

```
class CDerived : public CBasic
{
public:
    virtual int test(){ return 1;}
};

int main()
{
    CBasic       cBasic;
    CDerived    cDerived;
    CBasic * pB1 = new CBasic;
    CBasic * pB2 = new CDerived;
    CBasic * pB3 = new CBasic;
    CBasic * pB4 = new CDerived;


    //dynamic cast failed, so pD1 is null.
    CDerived * pD1 = dynamic_cast<CDerived * > (pB1);

    //dynamic cast succeeded, so pD2 points to  CDerived object

    CDerived * pD2 = dynamic_cast<CDerived * > (pB2);
    //pD3将是一个指向该CBasic类型对象的指针，对它进行CDerive类型的操作将是不安全的
    CDerived * pD3 = static_cast<CDerived * > (pB3);
    //static_cast成功
    CDerived * pD4 = static_cast<CDerived * > (pB4);

    //dynamci cast failed, so throw an exception.
//    CDerived & rD1 = dynamic_cast<CDerived &> (*pB1);

//dynamic cast succeeded, so rD2 references to CDerived object.
    CDerived & rD2 = dynamic_cast<CDerived &> (*pB2);

    return 0;
}
```

注：CBasic要有虚函数，否则会编译出错；**static_cast**则没有这个限制。

27、智能指针

1、智能指针是在 头文件中的std命名空间中定义的，该指针用于确保程序不存在内存和资源泄漏且是异常安全的。它们对**RAII"获取资源即初始化"**编程至关重要，RAII的主要原则是为**将任何堆分配资源（如动态分配内存或系统对象句柄）的所有权提供给其析构函数包含用于删除或释放资源的代码以及任何相关清理代码的堆栈分配对象。**大多数情况下，当初始化原始指针或资源句柄以指向实际资源时，会立即将指针传递给智能指针。

2、智能指针的设计思想：**将基本类型指针封装为类对象指针（这个类肯定是个模板，以适应不同基本类型的需求），并在析构函数里编写delete语句删除指针指向的内存空间。**

3、**unique_ptr**只允许基础指针的一个所有者。unique_ptr小巧高效；大小等同于一个指针且支持右值引用，从而可实现快速插入和对STL集合的检索。

4、**shared_ptr采用引用计数的智能指针，主要用于要将一个原始指针分配给多个所有者（例如，从容器返回了指针副本又想保留原始指针时）的情况。**当所有的shared_ptr所有者超出了范围或放弃所有权，才会删除原始指针。大小为两个指针；一个用于对象，另一个用于包含引用计数的共享控制块。最安全的分配和使用动态内存的方法是调用**make_shared**标准库函数，此函数在动态分配内存中分配一个对象并初始化它，返回对象的shared_ptr。

28、计算类大小例子

class A {};: sizeof(A) = 1; class A { virtual Fun(){} };: sizeof(A) = 4(32位机器)/8(64位机器); class A { static int a; };: sizeof(A) = 1; class A { int a; };: sizeof(A) = 4; class A { static int a; int b; };: sizeof(A) = 4;

类中用static声明的成员变量不计算入类的大小中，因为static data不是实例的一部分。static的属于全局的，他不会占用类的存储，他有专门的地方存储 （全局变量区）

29、大端与小端的概念？各自的优势是什么？

- 大端与小端是用来描述多字节数据在内存中的存放顺序，即字节序。**大端（Big Endian）**指低地址端存放高位字节，**小端（Little Endian）**是指低地址端存放低位字节。
- 需要记住计算机是**以字节为存储单位。**
- 为了方便记忆可把大端和小端称作高尾端和低尾端，eg：如果是高尾端模式一个字符串“11223344”把尾部“44”放在地址的高位，如果是地尾端模式，把“44”放在地址的低位。

各自优势：

- **Big Endian**：符号位的判定固定为第一个字节，容易判断正负。
- **Little Endian**：长度为1，2，4字节的数，排列方式都是一样的，数据类型转换非常方便。

举一个例子，比如数字0x12 34 56 78在内存中的表示形式为：

- 1)大端模式：

低地址 -----------------> 高地址 **0x12 | 0x34 | 0x56 | 0x78**

- 2)小端模式：

低地址 -----------------> 高地址 **0x78 | 0x56 | 0x34 | 0x12**

30、C++中*和&同时使用是什么意思?

template void InsertFront(Node* & head, T item) 上面一个函数的声明，其中第一个参数*和&分别是什么意思？ head是个指针，前面为什么加个&

本来“* head”代表的是传指针的，但是只能改变head指向的内容，而“* &head”意思是说head是传进来的指针的同名指针，就能既改变*head指向的内容，又能改变head这个指针。比如：main()有个Node* p,int t；当调用insertFront(p,t)是，如果template void InsertFront(Node* & head, T item)中有对head进行赋值改变时，main()中的p也会跟着改变，如果没有&这个别名标识时，p则不会随着head的改变而改变。

31、C++vector与list区别

参见：[https://www.cnblogs.com/shijingjing07/p/5587719.html

32、C语言中static关键字作用

在C语言中static的作用如下

第一、在修饰变量的时候，static修饰的静态局部变量只执行一次，而且延长了局部变量的生命周期，直到程序运行结束以后才释放。 第二、static修饰全局变量的时候，这个全局变量只能在本文件中访问，不能在其它文件中访问，即便是extern外部声明也不可以。 第三、static修饰一个函数，则这个函数的只能在本文件中调用，不能被其他文件调用。Static修饰的局部变量存放在全局数据区的静态变量区。初始化的时候自动初始化为0； （1）不想被释放的时候，可以使用static修饰。比如修饰函数中存放在栈空间的数组。如果不想让这个数组在函数调用结束释放可以使用static修饰 （2）考虑到数据安全性（当程想要使用全局变量的时候应该先考虑使用static)

在C++中static关键字除了具有C中的作用还有在类中的使用 在类中，static可以用来修饰静态数据成员和静态成员方法 静态数据成员 （1）静态数据成员可以实现多个对象之间的数据共享，它是类的所有对象的共享成员，它在内存中只占一份空间，如果改变它的值，则各对象中这个数据成员的值都被改变。 （2）静态数据成员是在程序开始运行时被分配空间，到程序结束之后才释放，只要类中指定了静态数据成员，即使不定义对象，也会为静态数据成员分配空间。 （3）静态数据成员可以被初始化，但是只能在类体外进行初始化，若为对静态数据成员赋初值，则编译器会自动为其初始化为0 （4）静态数据成员既可以通过对象名引用，也可以通过类名引用。

静态成员函数 （1）静态成员函数和静态数据成员一样，他们都属于类的静态成员，而不是对象成员。 （2）非静态成员函数有this指针，而静态成员函数没有this指针。 （3）静态成员函数主要用来访问静态数据成员而不能访问非静态成员。

33、C/C++中堆和栈的区别

讲解全面的一篇博客：https://blog.csdn.net/Fiorna0314/article/details/49757195

链接1

链接2

34、定义一个空类编译器做了哪些操作?

如果你只是声明一个空类，不做任何事情的话，编译器会自动为你生成一个默认构造函数、一个拷贝默认构造函数、一个默认拷贝赋值操作符和一个默认析构函数。这些函数只有在第一次被调用时，才会被编译器创建。所有这些函数都是inline和public的。

定义一个空类例如：

```
class Empty
{
}
```

一个空的class在C++编译器处理过后就不再为空，编译器会自动地为我们声明一些member function，一般编译过就相当于：

```
class Empty
{
public:
Empty(); // 缺省构造函数//
Empty( const Empty& ); // 拷贝构造函数//
~Empty(); // 析构函数//
Empty& operator=( const Empty& ); // 赋值运算符//
};
```

需要注意的是，只有当你需要用到这些函数的时候，编译器才会去定义它们。

35、友元函数和友元类

链接

36、什么情况下，类的析构函数应该声明为虚函数？为什么?

基类指针可以指向派生类的对象（多态性），如果删除该指针delete []p；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。

37、哪些函数不能成为虚函数?

不能被继承的函数和不能被重写的函数。

1、普通函数

普通函数不属于成员函数，是不能被继承的。普通函数只能被重载，不能被重写，因此声明为虚函数没有意义。因为编译器会在编译时绑定函数。

而多态体现在运行时绑定。通常通过基类指针指向子类对象实现多态。

2、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。

3、构造函数

首先说下什么是构造函数，构造函数是用来初始化对象的。假如子类可以继承基类构造函数，那么子类对象的构造将使用基类的构造函数，而基类构造函数并不知道子类的有什么成员，显然是不符合语义的。从另外一个角度来讲，多态是通过基类指针指向子类对象来实现多态的，在对象构造之前并没有对象产生，因此无法使用多态特性，这是矛盾的。因此构造函数不允许继承。

4、内联成员函数

我们需要知道内联函数就是为了在代码中直接展开，减少函数调用花费的代价。也就是说内联函数是在编译时展开的。而虚函数是为了实现多态，是在运行时绑定的。因此显然内联函数和多态的特性相违背。

5、静态成员函数

首先静态成员函数理论是可继承的。但是静态成员函数是编译时确定的，无法动态绑定，不支持多态，因此不能被重写，也就不能被声明为虚函数。

38、编写一个有构造函数，析构函数，赋值函数，和拷贝构造函数的String类

```
1
2  //.h
3
4  class String{
5     public:
6        String(const char* str);
7        String(const String &other);
8        ~String();
9        String & operator=(const String &other);
10    private:
11       char* m_data;
12 };
```

```
1  //.cpp
2  String::String(const char*str){
3     if(str==NULL){
4        m_data=new char[1];
5        *m_data='\0';
6     }
7     else{
8        int length=strlen(str);
9        m_data=new char[length+1];
10       strcpy(m_data,str);
11    }
12 }
13
14 String::String(const String &other){
15    int length=strlen(other.m_data);
16    m_data=new char[length+1];
17    strcpy(m_data,other.m_data);
18 }
19
20 String::~String(){
21    delete [] m_data;
22 }
```

```
23
24 String::String& operator=(const String & other){
25     if(&other==this)return *this;//检查自赋值
26     delete[]m_data;//释放原有的内存资源
27     int length=strlen(other.m_data);
28     m_data=new char[length+1];
29     strcpy(m_data,other.m_data);
30     return *this;//返回本对象的引用
31 }
```

注：一个单链表的简单实现：链接

39、this指针的理解

链接

40、程序加载时的内存分布（里面有linux中内存分区）

- 在多任务操作系统中，每个进程都运行在一个**属于自己的虚拟内存**中，而虚拟内存被分为许多页，并映射到物理内存中，被加载到物理内存中的文件才能够被执行。这里我们主要关注程序被装载后的内存布局，其可执行文件包含了代码段，数据段，BSS段，堆，栈等部分，其分布如下图所示。

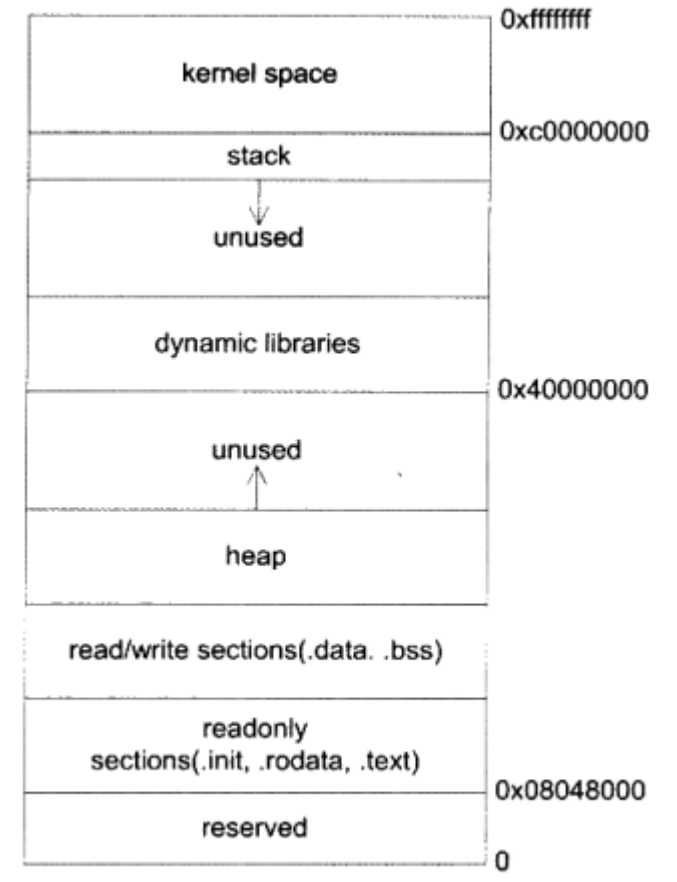


图 10-1　Linux 进程地址空间布局（内核版本 2.4.x）

- **代码段(.text)**：用来存放可执行文件的机器指令。存放在只读区域，以防止被修改。
- **只读数据段(.rodata)**：用来存放常量存放在只读区域，如字符串常量、全局const变量等。
- **可读写数据段(.data)**：用来存放可执行文件中已初始化全局变量，即静态分配的变量和全局变量。
- **BSS段(.bss)**：未初始化的全局变量和局部静态变量以及初始化为0的全局变量一般放在.bss的段里，以节省内存空间。eg:static int a=0;(初始化为0的全局变量（静态变量）放在.bss)。
- **堆**：用来容纳应用程序动态分配的内存区域。当程序使用malloc或new分配内存时，得到的内存来自堆。堆通常位于栈的下方。
- **栈**：用于维护函数调用的上下文。栈通常分配在用户空间的最高地址处分配。
- **动态链接库映射区**：如果程序调用了动态链接库，则会有这一部分。该区域是用于映射装载的动态链接库。

- **保留区**：内存中受到保护而禁止访问的内存区域。

41、智能指针

- 智能指针是在 头文件中的std命名空间中定义的，该指针用于确保程序不存在内存和资源泄漏且是异常安全的。它们对**RAII"获取资源即初始化"**编程至关重要，RAII的主要原则是为**将任何堆分配资源（如动态分配内存或系统对象句柄）的所有权提供给其析构函数包含用于删除或释放资源的代码以及任何相关清理代码的堆栈分配对象。**大多数情况下，当初始化原始指针或资源句柄以指向实际资源时，会立即将指针传递给智能指针。
- 智能指针的设计思想：**将基本类型指针封装为类对象指针（这个类肯定是个模板，以适应不同基本类型的需求），并在析构函数里编写delete语句删除指针指向的内存空间。**
- **unique_ptr**只允许基础指针的一个所有者。unique_ptr小巧高效；大小等同于一个指针且支持右值引用，从而可实现快速插入和对STL集合的检索。
- **shared_ptr采用引用计数的智能指针，主要用于要将一个原始指针分配给多个所有者（例如，从容器返回了指针副本又想保留原始指针时）的情况。**当所有的shared_ptr所有者超出了范围或放弃所有权，才会删除原始指针。大小为两个指针；一个用于对象，另一个用于包含引用计数的共享控制块。最安全的分配和使用动态内存的方法是调用**make_shared**标准库函数，此函数在动态分配内存中分配一个对象并初始化它，返回对象的shared_ptr。
- **注**：
  - **1. 引用计数问题**
    - 每个shared_ptr所指向的**对象**都有一个引用计数，它记录了有多少个shared_ptr指向自己
    - shared_ptr的析构函数：递减它所指向的对象的引用计数，如果引用计数变为0，就会销毁对象并释放相应的内存
    - 引用计数的变化：决定权在shared_ptr，而与对象本身无关
  - **2.智能指针支持的操作**
    - 使用重载的->和*运算符访问对象。
    - 使用**get成员函数获取原始指针**，提供对原始指针的直接访问。你可以使用智能指针管理你自己的代码中的内存，还能将原始指针传递给不支持智

能指针的代码。
- 使用删除器定义自己的释放操作。
- 使用**release**成员函数的作用是放弃智能指针对指针的控制权，将智能指针置空，并返回原始指针。（只支持unique_ptr）
- 使用**reset**释放智能指针对对象的所有权。

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

class base
{
public:
    base(int _a): a(_a)   {cout<<"构造函数"<<endl;}
    ~base()   {cout<<"析构函数"<<endl;}
    int a;
};

int main()
{
    unique_ptr<base> up1(new base(2));
    // unique_ptr<base> up2 = up1;   //编译器提示未定义
    unique_ptr<base> up2 = move(up1);  //转移对象的所有权
    // cout<<up1->a<<endl; //运行时错误
    cout<<up2->a<<endl; //通过解引用运算符获取封装的原始指针
    up2.reset(); // 显式释放内存

    shared_ptr<base> sp1(new base(3));
    shared_ptr<base> sp2 = sp1;  //增加引用计数
    cout<<"共享智能指针的数量："<<sp2.use_count()<<endl;  //2
    sp1.reset();  //
    cout<<"共享智能指针的数量："<<sp2.use_count()<<endl;  //1
    cout<<sp2->a<<endl;
    auto sp3 = make_shared<base>(4);//利用make_shared函数动态分配内存
}
```

3. 智能指针的陷阱（循环引用等问题）

链接

```
class B;
class A
{
public:
	shared_ptr<B> m_b;
};
class B
{
public:
	shared_ptr<A> m_a;
};

int main()
{
	{
		shared_ptr<A> a(new A);  //new出来的A的引用计数此时为1
		shared_ptr<B> b(new B);  //new出来的B的引用计数此时为1
		a->m_b = b;			//B的引用计数增加为2
		b->m_a = a;			//A的引用计数增加为2
	}
	//b先出作用域，B的引用计数减少为1，不为0；
	//所以堆上的B空间没有被释放，且B持有的A也没有机会被析构，A的引用计数也完全没减少

	//a后出作用域，同理A的引用计数减少为1，不为0，所以堆上A的空间也没有被释放
}
```

循环引用”简单来说就是：**两个对象互相使用一个shared_ptr成员变量指向对方会造成循环引用。**

即A内部有指向B，B内部有指向A，这样对于A，B必定是在A析构后B才析构，对于B，A必定是在B析构后才析构A，这就是循环引用问题，违反常规，导致内存泄露。

解决循环引用方法：

1. 当只剩下最后一个引用的时候需要手动打破循环引用释放对象。

2. 当A的生存期超过B的生存期的时候，B改为使用一个普通指针指向A。
3. 使用weak_ptr打破这种循环引用，因为weak_ptr不会修改计数器的大小，所以就不会产生两个对象互相使用一个shared_ptr成员变量指向对方的问题，从而不会引起引用循环。

42、vector扩容原理说明

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

1. vector在push_back以成倍增长可以在均摊后达到O(1)的事件复杂度，相对于增长指定大小的O(n)时间复杂度更好。
2. 为了防止申请内存的浪费，现在使用较多的有2倍与1.5倍的增长方式，而1.5倍的增长方式可以更好的实现对内存的重复利用，因为更好。

43、内联函数和宏定义的区别

1、**宏定义**不是函数，但是使用起来像函数。预处理器用复制宏代码的方式代替函数的调用，省去了函数压栈退栈过程，提高了效率。

内联函数本质上是一个函数，内联函数一般用于函数体的代码比较简单的函数，不能包含复杂的控制语句，while、switch，并且内联函数本身不能直接调用自身。如果内联函数的函数体过大，编译器会自动 的把这个内联函数变成普通函数。

2、**宏定义**是在**预处理**的时候把所有的宏名用宏体来替换，简单的说就是字符串替换

内联函数则是在**编译**的时候进行代码插入，编译器会在每处调用内联函数的地方直接把内联函数的内容展开，这样可以省去函数的调用的开销，提高效率

3、**宏定义**是没有类型检查的，无论对还是错都是直接替换

内联函数在编译的时候会进行类型的检查，内联函数满足函数的性质，比如有返回值、参数列表等

4、宏定义和内联函数使用的时候都是进行代码展开。不同的是**宏定义**是在预编译的时候把所有的宏名替换，**内联函数**则是在编译阶段把所有调用内联函数的地方把内联函数插入。这样可以省去函数压栈退栈，提高了效率

44、内联函数与普通函数的区别

1、内联函数和普通函数的参数传递机制相同，但是编译器会在每处调用内联函数的地方将内联函数内容展开，这样既避免了函数调用的开销又没有宏机制的缺陷。

2、普通函数在被调用的时候，系统首先要到函数的入口地址去执行函数体，执行完成之后再回到函数调用的地方继续执行，函数始终只有一个复制。

3、内联函数不需要寻址，当执行到内联函数的时候，将此函数展开，如果程序中有N次调用了内联函数则会有N次展开函数代码。

4、内联函数有一定的限制，内联函数体要求代码简单，不能包含复杂的结构控制语句。如果内联函数函数体过于复杂，编译器将自动把内联函数当成普通函数来执行。

45、C++中成员函数能够同时用static和const进行修饰?

不能。

C++编译器在实现const的成员函数（const加在函数右边）的时候为了确保该函数不能修改类的中参数的值，会在函数中添加一个隐式的参数const this*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。

即：static修饰的函数表示该函数是属于类的，而不是属于某一个对象的，没有this指针。const修饰的函数表示该函数不能改变this中的内容，会有一个隐含的const this指针。两者是矛盾的。

46、溢出，越界，泄漏

1.溢出

要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。

1）栈溢出

a.栈溢出是指函数中的局部变量造成的溢出（注：函数中形参和函数中的局部变量存放在栈上）

栈的大小通常是1M-2M,所以栈溢出包含两种情况，一是分配的的大小超过栈的最大值，二是分配的大小没有超过最大值，但是接收的buff比新buff小（buff：缓冲区，它本质上就是一段存储数据的内存）

例子1：（分配的的大小超过栈的最大值）

```
void
{
   char a[99999999999999999];
}
```

例子2：（接收的buff比新buff小）

```
void
{
  char a[10] = {0};
  strcpy(a, "abjjijjlljiojohihiihiiiiiiiiiiiiiiiiiiiiiii");
}
```

注意：调试时栈溢出的异常要在函数调用结束后才会检测到，因为栈是在函数结束时才会开始进行出栈操作

如：

```
int main(int argc, char* argv[])

{
char a[10] = {0};

strcpy(a, "abjjijjlljiojohihiihiiiiiiiiiiiiiiiiiiiiiii");

exit(0);

return 0;

}
```

上面情况是检测不到栈溢出的，因为函数还没执行完就退出了

```
void fun()
{
 char a[10] = {0};
 strcpy(a, "abjjijjlljiojohihiihiiiiiiiiiiiiiiiiiiiiiii");
}
int main(int argc, char* argv[])
{
  fun();
  exit(0);
  return 0;
}
```

这种情况调用完fun函数就会检测到异常了

b.栈溢出的解决办法

如果是超过栈的大小时，那就直接换成用堆；如果是不超过栈大小但是分配值小的，就增大分配的大小

2）内存溢出

使用malloc和new分配的内存，在拷贝时接收buff小于新buff时造成的现象

解决：增加分配的大小

2.越界

越界通常指的是数组越界，如

char a[9]={0};

cout << a[9] << endl;

3.泄漏

这里泄漏通常是指堆内存泄漏，是指使用malloc和new分配的内存没有释放造成的

47、C/C++中分配内存的方法

1、malloc 函数： void *malloc(unsigned int size)

在内存的动态分配区域中分配一个长度为size的连续空间，如果分配成功，则返回所分配内存空间的首地址，否则返回NULL，申请的内存不会进行初始化。

2、calloc 函数： void *calloc(unsigned int num, unsigned int size)

按照所给的数据个数和数据类型所占字节数，分配一个 num * size 连续的空间。

calloc申请内存空间后，会自动初始化内存空间为 0，但是malloc不会进行初始化，其内存空间存储的是一些随机数据。 3、realloc 函数： void *realloc(void *ptr, unsigned int size)

动态分配一个长度为size的内存空间，并把内存空间的首地址赋值给ptr，把ptr内存空间调整为size。

申请的内存空间不会进行初始化。 4、new是动态分配内存的运算符，自动计算需要分配的空间，在分配类类型的内存空间时，同时调用类的构造函数，对内存空间进行初始化，即完成类的初始化工作。动态分配内置类型是否自动初始化取决于变量定义的位置，在函数体外定义的变量都初始化为0，在函数体内定义的内置类型变量都不进行初始化。

48、构造函数初始化列表

构造函数初始化列表以一个冒号开始，接着是以逗号分隔的数据成员列表，每个数据成员后面跟一个放在括号中的初始化式。例如：

```
class CExample {
public:
   int a;
   float b;
   //构造函数初始化列表
   CExample(): a(0),b(8.8)
   {}
   //构造函数内部赋值
   CExample()
   {
      a=0;
      b=8.8;
   }
};
```

上面的例子中两个构造函数的结果是一样的。上面的构造函数（使用初始化列表的构造函数）显式的初始化类的成员；而没使用初始化列表的构造函数是对类的成员赋值，并没有进行显式的初始化。

初始化和赋值对内置类型的成员没有什么大的区别，像上面的任一个构造函数都可以。**对非内置类型成员变量，为了避免两次构造，**推荐使用类构造函数初始化列表**。但有的时候必须用带有初始化列表的构造函数： 1.成员类型是没有默认构造函数的类。若没有提供显示初始化式，则编译器隐式使用成员类型的默认构造函数，若类没有默认构造函数，则编译器尝试使用默认构造函数将会失败。 2.const成员或引用类型的成员。因为const对象或引用类型只能初始化，不能对他们赋值。

初始化数据成员与对数据成员赋值的含义是什么？有什么区别？ 首先把数据成员按类型分类并分情况说明: 1.**内置数据类型，复合类型（指针，引用）** 在成员初始化列表和构造函数体内进行，在性能和结果上都是一样的 2.**用户定义类型（类类型）** 结果上相同，但是**性能上存在很大的差别**。因为类类型的数据成员对象在进入函数体前已经构造完成（先进行了一次隐式的默认构造函数调用），也就是说在成员初始化列表处进行构造对象的工作，调用构造函数，在进入函数体之后，进行的是对已经构造好的类对象的赋值，又调用了拷贝赋值操作符才能完成（如果并未提供，则使用编译器提供的默认按成员赋值行为）。

49、vector中v[i]与v.at(i)的区别

```
void f(vector<int> &v)
{
  v[5];   // A
  v.at[5];  // B
}
```

如果v非空，A行和B行没有任何区别。如果v为空，B行会抛出std::out_of_range异常，A行的行为未定义。

c++标准不要求vector::operator[]进行下标越界检查，原因是为了效率，总是强制下标越界检查会增加程序的性能开销。设计vector是用来代替内置数组的，所以效率问题也应该考虑。不过使用operator[]就要自己承担越界风险了。

如果需要下标越界检查，请使用at。但是请注意，这时候的性能也是响应的会受影响，因为越界检查增加了性能的开销。

50、指向函数的指针--函数指针

链接

51、C++中调用C的函数

extern "C"

链接

52、指针常量与常量指针

常量指针(被指向的对象是常量)

定义：又叫常指针，可以理解为**常量的指针**，指向的是个常量

关键点：

1. 常量指针指向的对象不能通过这个指针来修改，可是仍然可以通过原来的声明修改；
2. 常量指针可以被赋值为变量的地址，之所以叫常量指针，是限制了通过这个指针修改变量的值；

3. 指针还可以指向别处，因为指针本身只是个变量，可以指向任意地址；

```
const int *p或int const *p
```

(记忆技巧：const读作常量，*读作指针)

```
#include <stdio.h>
// 常量指针(被指向的对象是常量)
int main() {
    int i = 10;
    int i2 = 11;
    const int *p = &i;
    printf("%d\n", *p);//10
    i = 9; //OK,仍然可以通过原来的声明修改值，
    //Error,*p是const int的，不可修改，即常量指针不可修改其指向地址
    //*p = 11;  //error: assignment of read-only location ‘*p’
    p = &i2;//OK,指针还可以指向别处，因为指针只是个变量，可以随意指向；
    printf("%d\n", *p);//11
    return 0;
}
```

指针常量(指针本身是常量)

定义：

本质是一个常量，而用指针修饰它。指针常量的值是指针，这个值因为是常量，所以不能被赋值。

关键点：

1. 它是个常量！
2. 指针所保存的地址可以改变，然而指针所指向的值却不可以改变；
3. 指针本身是常量，指向的地址不可以变化,但是指向的地址所对应的内容可以变化；

```
int* const p;
```

```
//指针常量(指针本身是常量)
#include <stdio.h>

```

```
int main() {
    int i = 10;
    int *const p = &i;
    printf("%d\n", *p);//10
    //Error,因为p是const 指针，因此不能改变p指向的内容
    //p++;//error: increment of read-only variable ‘p’
    (*p)++;   //OK,指针是常量，指向的地址不可以变化,但是指向的地址所对应的内容可以变化
    printf("%d\n", *p);//11
    i = 9;//OK,仍然可以通过原来的声明修改值，
    return 0;
}
```

53、防止头文件被重复包含

链接

54、详解拷贝构造函数相关知识

非常好的一篇博客：链接

55、为什么虚函数比普通函数慢?

虚函数在调用时需要通过查找虚函数表的方式来访问，故而比普通函数慢。

56、extern “C”

extern "C"是用来实现C和C++混合编译的。

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语言的进行编译，而不是C++的。 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。 假如，某个函数的原型为void foo(int x, int y);该函数被C编译器编译后在符号库中的名字为*foo，而C++编译器则会产生*foo_int_int之类的名字。_foo_int_int这样的名字是包含了函数名以及形参，C++就是靠这种机制来实现函数重载的。 **使用场景：**

a. C++代码调用C语言代码

b. 在C++的头文件中使用

c. 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

a. C++调用C语言代码：

```
/* c语言头文件：cExample.h */
        #ifndef C_EXAMPLE_H
        #define C_EXAMPLE_H
        extern int add(int x, int y);
        #endif

        /* c语言的实现文件：cExample.c */
        #include "cExample.h"
        int add(int x, int y)
        {
            return x + y;
        }

        /* c++实现文件，调用add：cppFile.cpp */
        extern "C"
        {
            #include "cExample.h";
        }
        int main()
        {
            add(2, 3);
            return 0;
        }

```

b. 在C++头文件中使用。

```
// C中引用C++语言中的函数或者变量时，C++的头文件需要加上extern “C”，


// 但是C语言中不能直接引用声明了extern “C”的该头文件，应该仅在C中将C++中定
```

```
义的extern “C”函数声明为extern类型。
    /* c++头文件cppExample.h */
    #ifndef CPP_EXAMPLE_H
    #define CPP_EXAMPLE_H
    extern "C" int add(int x, int y);
    #endif

    /* c实现文件cFile.c */
    extern int add(int x, int y);
    int main()
    {
        add(2, 3);
        return 0;
    }
```

57、结构体与类的区别

1、实例化的结构体存储在栈内，实例化的类存储在堆内，结构体的效率较高。

2、 结构体没有析构函数。

3、结构体不可以继承。

4、 结构体的默认成员类型为public，类为private。

面试常见问题

1、 extern关键字的作用

extern置于变量或函数前，用于标示变量或函数的定义在别的文件中，提示编译器遇到此变量和函数时在其他模块中寻找其定义。它只要有两个作用：

当它与“C”一起连用的时候，如：extern "C" void fun(int a,int b);则告诉编译器在编译fun这个函数时候按着C的规矩去翻译，而不是C++的（这与C++的重载有关，C++语言支持函数重载，C语言不支持函数重载，函数被C++编译器编译后在库中的名字与C语言的不同） 当extern不与“C”在一起修饰变量或函数时，如：extern int g_Int；它的作用就是声明函数或全局变量的作用范围的关键字，其声明的函数和变量可以在本模块或其他模块中使用。记住它是一个声明不是定义!也就是说B模块(编译单元)要是引用模块(编译单元)A中定义的全局变量或函数时，它只要包含A模块的头文件即可,在编译阶段，模块B虽然找不到该函数或变量，但它不会报错，它会在连接时从模块A生成的目标代码中找到此函数。

2、static关键字的作

修饰局部变量　static修饰局部变量时，使得被修饰的变量成为静态变量，存储在静态区。存储在静态区的数据生命周期与程序相同，在main函数之前初始化，在程序退出时销毁。（无论是局部静态还是全局静态）

修饰全局变量　全局变量本来就存储在静态区，因此static并不能改变其存储位置。但是，static限制了其链接属性。被static修饰的全局变量只能被该包含该定义的文件访问（即改变了作用域）。

修饰函数　static修饰函数使得函数只能在包含该函数定义的文件中被调用。对于静态函数，声明和定义需要放在同一个文件夹中。

修饰成员变量　用static修饰类的数据成员使其成为类的全局变量，会被类的所有对象共享，包括派生类的对象，所有的对象都只维持同一个实例。 因此，static成员必须在类外进行初始化(初始化格式：int base::var=10;)，而不能在构造函数内进行初始化，不过也可以用const修饰static数据成员在类内初始化。 修饰成员函数
用static修饰成员函数，使这个类只存在这一份函数，所有对象共享该函数，不含this指针，因而只能访问类的static成员变量。静态成员是可以独立访问的，也就是说，无须创建任何对象实例就可以访问。例如可以封装某些算法，比如数学函数，如ln，sin，tan等等，这些函数本就没必要属于任何一个对象，所以从类上调用感觉更好，比如定义一个数学函数类Math，调用Math::sin(3.14);还可以实现某些特殊的设计模式：如Singleton；

最重要的特性：隐藏

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，其它的源文件也能访问。利用这一特性可以在不同的文件中定义同名函数和同名变量，而不必担心命名冲突。static可以用作函数和变量的前缀，对于函数来讲，static的作用仅限于隐藏。

不可以同时用const和static修饰成员函数

C++编译器在实现const的成员函数的时候为了确保该函数不能修改类的实例的状态，会在函数中添加一个隐式的参数const this*。但当一个成员为static的时候，该函数是没有this指针的。也就是说此时const的用法和static是冲突的。我们也可以这样理解：两者的语意是矛盾的。static的作用是表示该函数只作用在类型的静态变量上，与类的实例没有关系；而const的作用是确保函数不能修改类的实例的状态，与类型的静态变量没有关系。因此不能同时用它们。

3、volatile的作用

用来修饰变量的，表明某个变量的值可能会随时被外部改变，因此这些变量的存取不能被缓存到寄存器，每次使用需要重新读取。

假如有一个对象A里面有一个boolean变量a，值为true,现在有两个线程T1，T2访问变量a，T1把a改成了false后T2读取a，T2这时读到的值可能不是false，即T1修改a的这一操作，对T2是不可见的。发生的原因可能是，针对T2线程，为了提升性能，虚拟机把a变量置入了寄存器（即C语言中的寄存器变量），这样就会导致，无论T2读取多少次a，a的值始终为true，因为T2读取了寄存器而非内存中的值。声明了volatile或synchronized 后，就可以保证可见性，确保T2始终从内存中读取变量，T1始终在内存中修改变量。总结：防止脏读，增加内存屏障。

也可参考该地址19问答案：http://www.sohu.com/a/166382914_538662

4、const的作用

定义变量为只读变量，不可修改 修饰函数的参数和返回值（后者应用比较少，一般为值传递） const成员函数（只需要在成员函数参数列表后加上关键字const，如char get() const;）可以访问const成员变量和非const成员变量，但不能修改任何变量。在声明一个成员函数时，若该成员函数并不对数据成员进行修改操作，应尽可能将该成员函数声明为const成员函数。 const对象只能访问const成员函数,而非const

对象可以访问任意的成员函数,包括const成员函数.即对于class A，有const A a；那么a只能访问A的const成员函数。而对于：A b；b可以访问任何成员函数。

使用const关键字修饰的变量，一定要对变量进行初始化

5、指针与引用的区别

1、指针只是一个变量，只不过这个变量存储的是一个地址；而引用跟原来的变量实质上是同一个东西，只不过是原变量的一个别名而已，不占用内存空间。 2、引用必须在定义的时候初始化，而且初始化后就不能再改变；而指针不必在定义的时候初始化，初始化后可以改变。 3、指针可以为空，但引用不能为空（这就意味着我们拿到一个引用的时候，是不需要判断引用是否为空的，而拿到一个指针的时候，我们则需要判断它是否为空。这点经常在判断函数参数是否有效的时候使用。） 4、“sizeof 引用" = 指向变量的大小， "sizeof 指针"= 指针本身的大小 5、指针可以有多级，而引用只能是一级

6、new与malloc的区别

1、malloc与free是C++/C语言的标准库函数，new/delete是C++的运算符。它们都可用于申请动态内存和释放内存。 2、对于非内部数据类型的对象而言，光用malloc/free无法满足动态对象的要求。对象在创建的同时要自动执行构造函数，对象在消亡之前要自动执行析构函数。 3、new可以认为是malloc加构造函数的执行。new出来的指针是直接带类型信息的。而malloc返回的都是void指针。

7、C++的多态性

多态性可以简单地概括为“一个接口，多种方法”，程序在运行时才决定调用的函数 。C++多态性主要是通过虚函数实现的，虚函数允许子类重写override(注意和overload的区别，overload是重载，是允许同名函数的表现，这些函数参数列表/类型不同)

多态与非多态的实质区别就是函数地址是早绑定还是晚绑定。如果函数的调用，在编译器编译期间就可以确定函数的调用地址，并生产代码，是静态的，就是说地址是早绑定的。而如果函数调用的地址不能在编译器期间确定，需要在运行时才确定，这就属于晚绑定。

在绝大多数情况下，程序的功能是在编译的时候就确定下来的，我们称之为静态特性。反之，如果程序的功能是在运行时刻才能确定下来的，则称之为动态特性。C++中，虚函数，抽象基类，动态绑定和多态构成了出色的动态特性。

最常见的用法就是声明基类的指针，利用该指针指向任意一个子类对象，调用相应的虚函数，可以根据指向的子类的不同而实现不同的方法。

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

有关重载，重写，覆盖的区别请移步：https://www.cnblogs.com/LUO77/p/5771237.html

8、虚函数表

请移步：https://www.cnblogs.com/LUO77/p/5771237.html

9、动态绑定与静态绑定

1、静态绑定发生在编译期，动态绑定发生在运行期； 2、对象的动态类型可以更改，但是静态类型无法更改； 3、要想实现动态，必须使用动态绑定； 4、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； 5、静态多态是指通过模板技术或者函数重载技术实现的多态，其在编译器确定行为。动态多态是指通过虚函数技术实现在运行期动态绑定的技术 动态绑定：有一个基类，两个派生类，基类有一个virtual函数，两个派生类都覆盖了这个虚函数。现在有一个基类的指针或者引用，当该基类指针或者引用指向不同的派生类对象时，调用该虚函数，那么最终调用的是该被指向对象对应的派生类自己实现的虚函数。

10、虚函数表是针对类的还是针对对象的？同一个类的两个对象的虚函数表是怎么维护的？

编译器为每一个类维护一个虚函数表（本质是一个函数指针数组，数组里面存放了一系列函数地址），每个对象的首地址保存着该虚函数表的指针，同一个类的不同对象实际上指向同一张虚函数表。调用形式：*(this指针+调整量)虚函数在vftable内的偏移

在类内部添加一个虚拟函数表指针，该指针指向一个虚拟函数表，该虚拟函数表包含了所有的虚拟函数的入口地址，每个类的虚拟函数表都不一样，在运行阶段可以循此脉络找到自己的函数入口。纯虚函数相当于占位符， 先在虚函数表中占一个位置由派生类实现后再把真正的函数指针填进去。除此之外和普通的虚函数没什么区别。

在单继承形式下，子类的完全获得父类的虚函数表和数据。子类如果重写了父类的虚函数（如fun），就会把虚函数表原本fun对应的记录（内容MyClass::fun）覆盖为新的函数地址（内容MyClassA::fun），否则继续保持原本的函数地址记录。

使用这种方式，就可以实现多态的特性。假设我们使用如下语句：

```
MyClass*pc=new MyClassA;
pc->fun();
```

因为虚函数表内的函数地址已经被子类重写的fun函数地址覆盖了，因此该处调用的函数正是MyClassA::fun，而不是基类的MyClass::fun。

如果使用MyClassA对象直接访问fun，则不会出发多态机制，因为这个函数调用在编译时期是可以确定的，编译器只需要直接调用MyClassA::fun即可。

注：对象不包含虚函数表，只有虚指针，类才包含虚函数表，派生类会生成一个兼容基类的虚函数表

详情可以参考：http://www.cnblogs.com/fanzhidongyzby/archive/2013/01/14/2859064.html

11、智能指针怎么实现？什么时候改变引用计数?

- 构造函数中计数初始化为1；
- 拷贝构造函数中计数值加1；
- 赋值运算符中，左边的对象引用计数减一，右边的对象引用计数加一；
- 析构函数中引用计数减一；
- 在赋值运算符和析构函数中，如果减一后为0，则调用delete释放对象。

12、内联函数，宏定义和普通函数的区别

1、内联函数要做参数类型检查，这是内联函数跟宏相比的优势 2、宏定义是在预编译的时候把所有的宏名用宏体来替换，简单的说就是字符串替换， 内联函数则是在编译的时候进行代码插入，编译器会在每处调用内联函数的地方直接把内联函数的内容展开，这样可以省去函数的调用的压栈出栈的开销，提高效率。 3、内联函数是指嵌入代码，就是在调用函数的地方不是跳转，而是把代码直接写到那里去。对于短小简单的代码来说，内联函数可以带来一定的效率提升，而且和C时代的宏函数相比，内联函数 更安全可靠。可是这个是以增加空间消耗为代价的 4、inline与#define的区别：宏在预处理阶段替换，inline在编译阶段替换；宏没有类型，不做安全检查，inline有类型，在编译阶段进行安全检查

13、C++内存管理

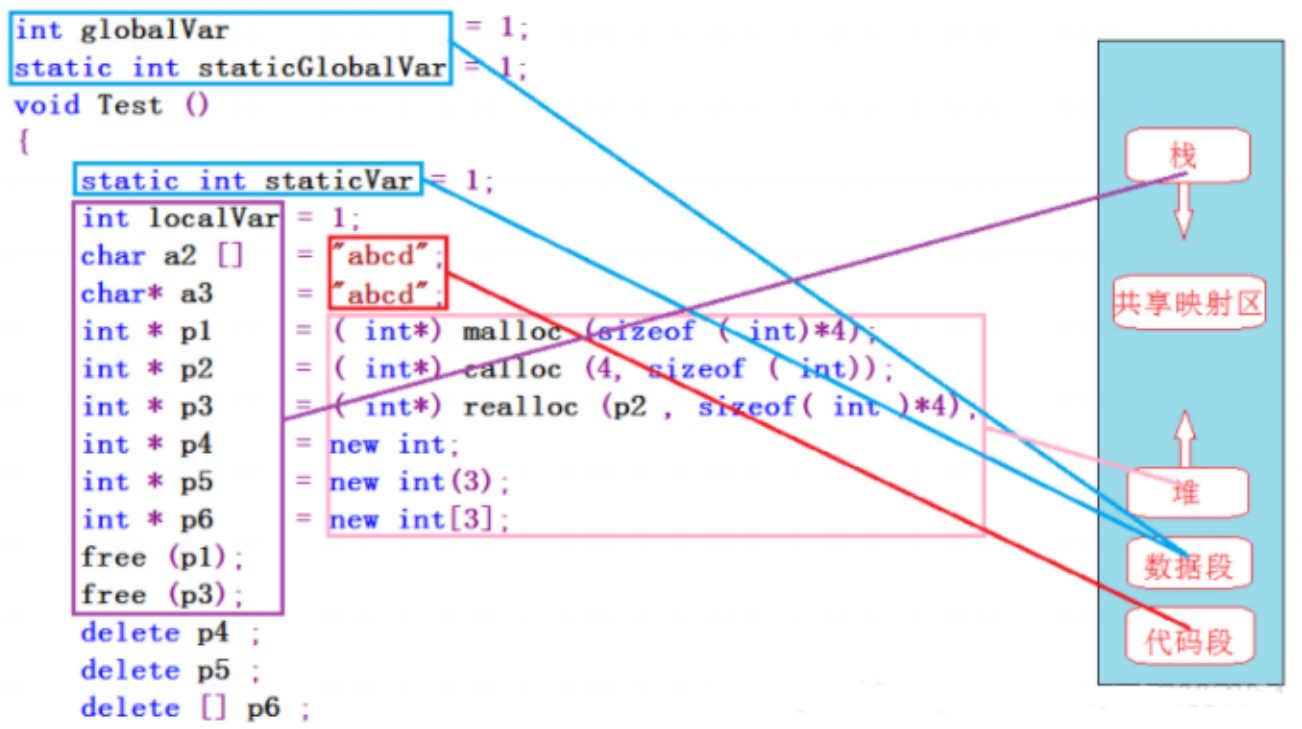
栈: 存放函数参数以及局部变量 , 在出作用域时 , 将自动被释放 . 栈内存分配运算内置于处理器的指令集中 , 效率 很 高 , 但分配的内存容量有限 .

堆 :new 分配的内存块 (包括数组 , 类实例等), 需 delete 手动释放 . 如果未释放 , 在整个程序结束后 ,OS 会帮你回收掉 .

自由存储区: malloc 分配的内存块 , 需 free 手动释放 . 它和堆有些相似 .

全局/静态区: 保存自动全局变量和static变量（包括static全局和局部变量）。静态区的内容在整个程序的生命周期内都存在，有编译器在编译的时候分配（数据段（存储全局数据和静态数据）和代码段（可执行的代码/只读常量））。

常量存储区: 常量 (const) 存于此处 , 此存储区不可修改 .



栈与堆的区别:

管理方式不同: 栈是编译器自动管理的,堆需手动释放 空间大小不同: 在32位OS下,堆内存可达到4GB的的空间,而栈就小得可怜.(VC6中,栈默认大小是1M,当然,你可以修改它) 能否产生碎片不同:对于栈来说,进栈/出栈都有着严格的顺序(先进后出),不会产生碎片;而堆频繁的new/delete,会造成内存空间的不连续,容易产生碎片. 生长方向不同:栈向下生长,以降序分配内存地址;堆向上生长,以升序分配内在地址. 分配方式不同:堆动态分配,无静态分配;栈分为静态分配和动态分配,比如局部变量的分配,就是动态分配(alloca函数),函数参数的分配就是动态分配(我想的...). 分配效率不同:栈是系统提供的数据结构,计算机会在底层对栈提供支持,进栈/出栈都有专门的指令,这就决定了栈的效率比较高.堆则不然,它由C/C++函数库提供,机制复杂,堆的效率要比栈低得多. 可以看出,栈的效率要比堆高很多,所以,推荐大家尽量用栈.不过,虽然栈有如此多的好处,但远没有堆使用灵活.

14、常用的设计模式

单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点；

工厂模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类。

参考：https://www.cnblogs.com/Y1Focus/p/6707121.html

https://www.zhihu.com/question/34574154?sort=created

15、手写strcpy，memcpy，strcat，strcmp等函数

https://blog.csdn.net/gao1440156051/article/details/51496782

https://blog.csdn.net/wilsonboliu/article/details/7919773

16、i++是否为原子操作?

不是。操作系统原子操作是不可分割的，在执行完毕不会被任何其它任务或事件中断，分为两种情况（两种都应该满足）

（1） 在单线程中， 能够在单条指令中完成的操作都可以认为是" 原子操作"，因为中断只能发生于指令之间。

（2） 在多线程中，不能被其它进程（线程）打断的操作就叫原子操作。

i++分为三个阶段：

1、内存到寄存器

2、寄存器自增

3、写回内存

这三个阶段中间都可以被中断分离开.

17、有关数组，指针，函数的三者结合问题

数组指针和指针数组的区别：https://blog.csdn.net/men_wen/article/details/52694069

右左法则的说明：http://www.cnblogs.com/zhangjing0502/archive/2012/06/08/2542059.html

指针常量和常量指针：https://www.zhihu.com/question/19829354

https://blog.csdn.net/xingjiarong/article/details/47282563

注意：所谓指向常量的指针或引用（即常量引用、常量指针），不过是指针或引用“自以为是”罢了，它们觉得自己指向了常量，所以自觉地不去改变所指对象的值，但这些对象却可以通过其他途径改变。

const int *a; 等价于int const *a; const在前面所以内容不可以改变，但是指针指向可以改变。也就是常量指针

int *const a; 表示的是指针指向不可改变，但是指针所存放的内容可以改变，也即是指针常量

18、C++中类与结构体的区别?

1、最本质的一个区别就是默认的访问控制： struct作为数据结构的实现体，它默认的数据访问控制是public的，而class作为对象的实现体，它默认的成员变量访问控制是private的。 2、“class”这个关键字还用于定义模板参数，就像“typename”。但关键字“struct”不用于定义模板参数。

19、析构函数的作用?

析构函数是用来释放所定义的对象中使用的指针，默认的析构函数不用显示调用，自建的析构函数要在程序末尾调用。

如果你的类里面只用到的基本类型，如int char double等，系统的默认析构函数其实什么都没有做

但如果你使用了其他的类如vector，string等，系统的默认析构函数就会调用这些类对象的析构函数

如果是自己写析构函数的话，如果你的类里面分配了系统资源，如new了内存空间，打开了文件等，那么在你的析构函数中就必须释放相应的内存空间和关闭相关的文件；这样系统就会自动调用你的析构函数释放资源，避免内存泄漏

例如：

```
1  class A
2  {
3  private:
4      char *data;

5  public:
```

```
    A()
    {
        data = new char[10];
    }
    ~A()
    {
        delete[] data;
    }
};
```

A a; a 中将 new 10个 char 当 a 这个变量消亡的时候，将自动执行 ~A()，释放空间

对象消亡时，自动被调用，用来释放对象占用的空间，避免内存泄漏

20、虚函数的作用?

虚函数可以让成员函数操作一般化，用基类的指针指向不同的派生类的对象时，基类指针调用其虚成员函数，则会调用其真正指向对象的成员函数，而不是基类中定义的成员函数（只要派生类改写了该成员函数）。若不是虚函数，则不管基类指针指向的哪个派生类对象，调用时都会调用基类中定义的那个函数。虚函数是C++多态的一种表现，可以进行灵活的动态绑定。 重点可参考：https://www.cnblogs.com/wangxiaobao/p/5850949.html

http://www.cnblogs.com/fanzhidongyzby/archive/2013/01/14/2859064.html

21、操作系统和编译器如何区分全局变量和局部变量?

操作系统只管调度进程，编译器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。局部变量则分配在栈里面 。

22、Makefile文件的作用?

makefile关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。

23、结构体和联合体的区别?

1、结构和联合都是由多个不同的数据类型成员组成, 但在任何同一时刻, 联合中只存放了一个被选中的成员（所有成员共用一块地址空间）, 而结构的所有成员都存在（不同成员的存放地址不同）。

2、对于联合的不同成员赋值, 将会对其它成员重写, 原来成员的值就不存在了, 而对于结构的不同成员赋值是互不影响的。

24、列表初始化问题?

使用初始化列表主要是基于性能问题，对于内置类型，如int, float等，使用初始化列表和在构造函数体内初始化差别不是很大；但是对于类类型来说，最好使用初始化列表。这样就可以直接调用拷贝构造函数初始化，省去了一次调用默认构造函数的过程。

```
struct Test1
{
   Test1() // 无参构造函数
   {
      cout << "Construct Test1" << endl ;
   }

   Test1(const Test1& t1)  // 拷贝构造函数
   {
      cout << "Copy constructor for Test1" << endl ;
      this->a = t1.a ;
   }

   Test1& operator = (const Test1& t1) // 赋值运算符
   {
      cout << "assignment for Test1" << endl ;
      this->a = t1.a ;
      return *this;
   }

   int a ;
};


struct Test2        //普通初始化
```

```
{
   Test1 test1 ;
   Test2(Test1 &t1)
   {
      test1 = t1 ;
   }
};
================================================================
=======
struct Test2        //2.列表初始化
{
   Test1 test1 ;
   Test2(Test1 &t1):test1(t1){}
}
================================================================
========
Test1 t1 ;          //调用
Test2 t2(t1) ;

```

普通初始化：

```
Construct Test1
Construct Test1
assignment for Test1
```

列表初始化：

```
Construct Test1
Copy constructor for Test1
```

下列情况一定要使用初始化成员列表

常量成员，因为常量只能初始化不能赋值，所以必须放在初始化列表里面 引用类型，引用必须在定义的时候初始化，并且不能重新赋值，所以也要写在初始化列表里面 需要初始化的数据成员是对象的情况　参考地址：https://www.cnblogs.com/weizhixiang/p/6374430.html

25、重载与重写的区别?

从定义上来说：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）。重写：是指子类重新定义父类虚函数的方法。

从实现原理上来说：重载：编译器根据函数不同的参数表，对同名函数的名称做修饰，然后这些同名函数就成了不同的函数。重写：当子类重新定义了父类的虚函数后，父类指针根据赋给它的不同的子类指针，动态的调用属于子类的该函数，这样的函数调用在编译期间是无法确定的（调用的子类的虚函数的地址无法给出）。

补充："隐藏"是指派生类的函数屏蔽了与其同名的基类函数。规则如下：　（1）如果派生类的函数与基类的函数同名，但是参数不同。此时，不论有无virtual关键字，基类的函数将被隐藏（注意别与重载混淆）。　（2）如果派生类的函数与基类的函数同名，并且参数也相同，但是基类函数没有virtual 关键字。此时，基类的函数被隐藏（注意别与覆盖混淆）。

26、类型安全以及C++中的类型转换?

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。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。

详情可以移步：https://blog.csdn.net/chengonghao/article/details/50974022

四种类型转换：

static_cast <T*> (content) 静态转换.在编译期间处理，可以实现C++中内置基本数据类型之间的相互转换。如果涉及到类的话，static_cast只能在有相互联系的类型中进行相互转换,不一定包含虚函数。 dynamic_cast<T*>(content) 动态类型转换;也是向下安全转型;是在运行的时候执行;基类中一定要有虚函数，否则编译不通过。在类层次间进行上行转换时（如派生类指针转为基类指针），dynamic_cast和static_cast的效果是一样的。在进行下行转换时（如基类指针转为派生类指针），dynamic_cast具有类型检查的功能，比static_cast更安全。 const_cast<T*> (content) 去常转换;编译时执行; reinterpret_cast<T*>(content) 重解释类型转换; 详情可以移步：https://blog.csdn.net/u010025211/article/details/48626687

https://blog.csdn.net/xtzmm1215/article/details/46475565

https://blog.csdn.net/xingkongfenqi/article/details/49148885

27、内存对齐的原则以及作用?

- 结构体内的成员按自身长度自对齐（32位机器上，如char=1，short=2，int=4，double=8），所谓自对齐是指该成员的起始地址必须是它自身长度的整数倍。如int只能以0,4,8这类地址开始。
- 结构体的总大小为结构体的有效对齐值的整数倍（默认以结构体中最长的成员长度为有效值的整数倍，当用#pragrma pack（n）指定时，以n和结构体中最长的成员的长度中较小者为其值）。即sizeof的值，必须是其内部最大成员的整数倍，不足的要补齐。

例如：

```
class A
{
   char c;
   int a;
   char d;
};

cout << sizeof(A) << endl;

class B
{
   char c;
   char d;
   int a;
};

cout << sizeof(B) << endl;
```

sizeof（A）=12，sizeof（B）=8；

因为左边是1+（3）+4+1+（3）=12，而右边是1+1+（2）+4=8。括号中为补齐的字节。

内存对齐的作用：

1、平台原因(移植原因)：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。

2、性能原因：经过内存对齐后，CPU的内存访问速度大大提升。

详情可以移步：https://blog.csdn.net/chy19911123/article/details/48894579

28、关键字registr，typdef的作用?

register关键字的作用：

请求CPU尽可能让变量的值保存在CPU内部的寄存器中，减去CPU从内存中抓取数据的时间，提高程序运行效率。

使用register关键字应注意什么?

1、只有局部变量才可以被声明用register修饰

（register不能修饰全局变量和函数的原因：全局变量可能被多个进程访问，而用register修饰的变量，只能被当前进程访问）

2、不能用取地址获取用register修饰的变量的地址（原因：变量保存在寄存器中，而取地址获取的地址的是内存的地址）

3、用register修饰的变量一定要是CPU所接受的数据类型

typedef关键字的作用：

给数据类型定义一个新名字，

1. 提高了移植性
2. 简化复杂的类型声明，提高编码效率
3. 解释数据类型的作用

29、什么情况下需要将析构函数定义为虚函数?

当基类指针指向派生类的对象（多态性）时。如果定义为虚函数，则就会先调用该指针指向的派生类析构函数，然后派生类的析构函数再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。所以，将析构函数声明为虚函数是十分必要的。

详情可以移步：https://blog.csdn.net/jiadebin890724/article/details/7951461

30、有关纯虚函数的理解?

纯虚函数是为你的程序制定一种标准，纯虚函数只是一个接口，是个函数的声明而已，它要留到子类里去实现。

```
class A{
protected:
void foo();//普通类函数
virtual void foo1();//虚函数
virtual void foo2() = 0;//纯虚函数
}
```

带纯虚函数的类抽象类，这种类不能直接生成对象，而只有被继承，并重写其虚函数后，才能使用。　虚函数是为了继承接口和默认行为　纯虚函数只是继承接口，行为必须重新定义

（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。所以引入了纯虚函数的概念）

详情可以参考：https://blog.csdn.net/ybhjx/article/details/51788396

31、基类指针指向派生类，派生类指针指向基类?

基类指针可以指向派生类对象，从而实现多态，例如：

```
#include <iostream>
using namespace std;
class Shape {
public:
   virtual double area() const = 0; //纯虚函数
};
 class Square : public Shape {
   double size;
public:
   Square(double s) {
      size = s;
```

```
    }
    virtual double area() const {
        return size * size;
    }
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) {
        radius = r;
    }
    virtual double area() const {
        return 3.14159 * radius * radius;
    }
};
 int main()
 {
     Shape* array[2]; //定义基类指针数组
     Square Sq(2.0);
     Circle Cir(1.0);
    array[0] = &Sq;
    array[1] =&Cir;
    for (int i = 0; i < 2; i++) /
    {
        cout << array[i]->area() << endl;
    }
    return 0;
}
```

上面的不同对象Sq,Cir(来自继承同一基类的不同派生类)接受同一消息（求面积，来自基类的成员函数area()）,但是却根据自身情况调用不同的面积公式（执行了不同的行为，它是通过虚函数实现的）。我们可以理解为，继承同一基类的不同派生对象，对来自基类的同一消息执行了不同的行为，这就是多态，它是通过继承和虚函数实现的。而接受同一消息的实现就是基于基类指针。

但是要注意的是，这个指针只能用来调用基类的成员函数。

如果试图通过基类指针调用派生类才有的成员函数，则编译器会报错。

为了避免这种错误，必须将基类指针强制转化为派生类指针。然后派生类指针可以用来调用派生类的功能。这称为向下强制类型转换，这是一种潜在的危险操作。

派生类指针不可以指向基类对象,例如：

有个people类是基类，成员有姓名和身份证号，有个派生类学生student，添加了成员学号，现在如果你说的这个情况成立student的指针----pt让他指向people成员t，则t只有两个成员变量，而*pt有3个，现在pt->学号这个变量在pt下是可以使用的，但它指向的实体却没有这个变量，所以出错，于是C++直接就避免了这样的隐式转换。 所以根据上述信息我们可以知道：

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）是不安全的。

参考链接：https://blog.csdn.net/flyingbird_sxf/article/details/41358737

https://www.cnblogs.com/rednodel/p/5800142.html

32、继承机制中引用和指针之间如何转换？ dynamic_cast

基类——>派生类：用dynamic_cast转换（显示转换），首先检查基类指针（引用）是否真正指向一个派生类对象，然后再做相应处理，对指针进行dynamic_cast，成功返回派生类对象，失败返回空指针，对引用进行dynamic_cast，成功返回派生类对象，失败抛出一个异常。 不允许隐式转换。 派生类——>基类：可以用dynamic_cast或者直接进行类型转换（直接赋值）。

33、c语言和c++有什么区别?

C语言是结构化的编程语言，它是面向过程的，而C++是面向对象的。 封装：将数据和函数等集合在一个单元中（即类）。被封装的类通常称为抽象数据类型。封装的意义在于保护或者防止代码（数据）被我们无意中破坏。 继承：继承主要实现重用代码，节省开发时间。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。 多态：同一操作作用于不同的对象，可以有不同的解释，产生不同的执行结果。在运行时，可以通过指向派生类的基类指针，来调用实现派生类中的方法。有编译时多态和运行时多态。

34、C++中的公有，私有，保护的问题?

	类	类对象	公有继承派生类	私有继承派生类	保护继承派生类	公有继承派生类对象	私有继承派生类对象	保护继承派生类对象
公有成员	√	√	√	√	√	√	X	X
私有成员	√	X	X	X	X	X	X	X
保护成员	√	X	√	√	√	X	X	X

√：代表可以访问，X代表不能访问。

参考链接：https://zhidao.baidu.com/question/551075894.html

35、如何实现类对象只能静态分配或动态分配?

C++中建立类的对象有两种方式：（1）静态建立，例如 A a;　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。使用这种方法，是直接调用类的构造函数。（2）动态建立，例如 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，在堆空间中搜索一块内存并进行分配；第二步调用类的构造函数构造对象。这种方法是间接调用类的构造函数。 **只能动态分配**：

其实，编译器在为类对象分配栈空间时，会先检查类的析构函数的访问性（其实不光是析构函数，只要是非静态的函数，编译器都会进行检查）。如果类的析构函数在类外部无法访问，则编译器拒绝在栈空间上为类对象分配内存。 因此，可以将析构函数设为private，这样就无法在栈上建立类对象了。但是为了子类可以继承，最好设置成protected。

```
class A
{
protected:
    A(){}
    ~A(){}
public:
    static A* create(){return new A();}
    void destory(){delete this;}
};
```

只能静态分配：

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。

```
class A
{
private:
    void* operator new(size_t t){}          //注意函数的第一个参数和返回值都是固定的
    void  operator delete(void* ptr)()      //重载了new就需要重载delete
public:
    A(){}
    ~A(){}
};
```

36、explicit关键字的作用?

C++中， 一个参数的 构造函数(或者除了第一个参数外其余参数都有默认值的多参构造函数)， 承担了两个角色。 1 是个 构造器 ， 2 是个默认且隐含的类型转换操作符。所以， 有时候在我们写下如 AAA = XXX， 这样的代码， 且恰好XXX的类型正好是AAA单参数构造器的参数类型， 这时候 编译器就自动调用这个构造器， 创建一个AAA的对象。

这样看起来好象很酷， 很方便。 但在某些情况下（见下面权威的例子）， 却违背了我们（程序员）的本意。 这时候就要在这个构造器前面加上explicit修饰， 指定这个构造器只能被明确的调用/使用， 不能作为类型转换操作符被隐含的使用。

```
1  class Test1
2  {
3  public:
4     Test1(int n)
5     {
6        num=n;
7     }//普通构造函数
8  private:
9     int num;
10 };
11 class Test2
12 {
13 public:
14    explicit Test2(int n)
15    {
16       num=n;
17    }//explicit(显式)构造函数
18 private:
19    int num;
20 };
21 int main()
22 {
23    Test1 t1=12;//隐式调用其构造函数,成功
24    Test2 t2=12;//编译错误,不能隐式调用其构造函数
25    Test2 t2(12);//显式调用成功
26    return 0;
27 }
```

Test1的 构造函数带一个int型的参数，代码23行会隐式转换成调用Test1的这个构造函数。而Test2的构造函数被声明为explicit（显式），这表示不能通过隐式转换来调用这个构造函数，因此代码24行会出现编译错误。 普通构造函数能够被 隐式调用。而explicit构造函数只能被显式调用。

37、内存溢出，内存泄漏的原因?

内存溢出是指程序在申请内存时，没有足够的内存空间供其使用。原因可能如下：

- 内存中加载的数据量过于庞大，如一次从数据库取出过多数据

- 代码中存在死循环或循环产生过多重复的对象实体
- 递归调用太深，导致堆栈溢出等
- 内存泄漏最终导致内存溢出

内存泄漏是指向系统申请分配内存进行使用（new），但是用完后不归还（delete），导致占用有效内存。常见的几种情况：

（1） 在类的构造函数和析构函数中没有匹配的调用new和delete函数

两种情况下会出现这种内存泄露：一是在堆里创建了对象占用了内存，但是没有显示地释放对象占用的内存；二是在类的构造函数中动态的分配了内存，但是在析构函数中没有释放内存或者没有正确的释放内存

（2） 在释放对象数组时在delete中没有使用方括号

方括号是告诉编译器这个指针指向的是一个对象数组，同时也告诉编译器正确的对象地址值病调用对象的析构函数，如果没有方括号，那么这个指针就被默认为只指向一个对象，对象数组中的其他对象的析构函数就不会被调用，结果造成了内存泄露。

（3）没有将基类的析构函数定义为虚函数

当基类指针指向子类对象时，如果基类的析构函数不是virtual，那么子类的析构函数将不会被调用，子类的资源没有正确是释放，因此造成内存泄露

参考链接： https://blog.csdn.net/hyqwmxsh/article/details/52813307

缓冲区溢出（栈溢出） 程序为了临时存取数据的需要，一般会分配一些内存空间称为缓冲区。如果向缓冲区中写入缓冲区无法容纳的数据，机会造成缓冲区以外的存储单元被改写，称为缓冲区溢出。而栈溢出是缓冲区溢出的一种，原理也是相同的。分为上溢出和下溢出。其中，上溢出是指栈满而又向其增加新的数据，导致数据溢出；下溢出是指空栈而又进行删除操作等，导致空间溢出。

38、auto_ptr类与shared_ptr类?

从c++11开始, auto_ptr已经被标记为弃用, 常见的替代品为shared_ptr。shared_ptr的不同之处在于引用计数, 在复制(或赋值)时不会像auto_ptr那样直接转移所有权。 两者都是模板类，却可以像指针一样去使用。只是在指针上面的一层封装。

auto_ptr实际也是一种类, 拥有自己的析构函数, 生命周期结束时能自动释放资源，正因为能自动释放资源, 特别适合在单个函数内代替new/delete的调用, 不用自己调用delete，也不用担心意外退出造成内存的泄漏。

atuo_ptr的缺陷:

- auto_ptr不能共享所有权，即不要让两个auto_ptr指向同一个对象（因为它采用的是转移语义的拷贝，原指针会变为NULL）。
- auto_ptr不能管理对象数组（因为它内部的析构函数调用的是delete而不是delete[]）。
- auto_ptr不能作为容器对象，STL容器中的元素经常要支持拷贝，赋值等操作，在这过程中auto_ptr会传递所有权。

详情原因可以参考：https://blog.csdn.net/uestclr/article/details/51316001

https://blog.csdn.net/kezunhai/article/details/38514823

shared_ptr 使用引用计数的方式来实现对指针资源的管理。同一个指针资源，可以被多个 shared_ptr 对象所拥有，直到最后一个 shared_ptr 对象析构时才释放所管理的对象资源。

可以说，shared_ptr 是最智能的智能指针，因为其特点最接近原始的指针。不仅能够自由的赋值和拷贝，而且可以安全的用在标准容器中。

39、有4种情况，编译器必须为未声明的constructor的classes合成一个default constructor：

- “带有默认构造函数”的成员对象
- “带有默认构造函数”的基类
- “带有虚函数”的类
- “带有虚拟基类”的类

被合成的构造函数只能满足编译器（而非程序员）的需要。在合成默认的构造函数中，只有基类的子对象和成员对象会被初始化，其他非静态的数据成员（如整数，指针等）都不会被初始化。

所以并不是任何的类如果没有定义默认的构造函数，都会被合成一个出来。

40、虚基类

在C++中,如果在多条继承路径上有一个公共的基类,那么在这些路径中的某几条路径的汇合处,这个公共的基类就会产生多个实例（从而造成二义性）.如果想使这个公共的基类只产生一个实例,则可将这个基类说明为虚基类. 这要求在从base类派生新类时,使用关键字virtual将base类说明为虚基类.

用例子说明吧。

```
class base{protected:int b};
clase base1:public base{..};
clase base2:public base{..};
clase derived:public base1,public base2 {..};
derived d;
d.b //错误.
d.base::b //错误. 因为不知是用d.base1::b还是d.base2::b
================================================
class base{protected:int b..};
clase base1:virtual public base{..}; //说明base为base1虚基类
clase base2:virtual public base{..}; //说明base为base2虚基类
clase derived:public base1,public base2 {..};
derived d;
d.b //对.
d.base::b //对. 因为d.base::b和d.base1::b还是d.base2::b都是引用同一虚基类成员b,具有相同的值.
```

41、模板的特例化

引入原因：编写单一的模板，它能适应大众化，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**：是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。

函数模板特例化：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。

```
template <typename T>
```

```
void fun(T a)
{
    cout << "The main template fun(): " << a << endl;
}

template <>   // 对int型特例化
void fun(int a)
{
    cout << "Specialized template for int type: " << a << endl;
}

int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(9.15);
    return 0;
}
```

对于除int型外的其他数据类型，都会调用通用版本的函数模板fun(T a)；对于int型，则会调用特例化版本的fun(int a)。注意，**一个特例化版本的本质是一个实例**，而非函数的重载。因此，特例化不影响函数匹配。

类模板的特例化：

```
template <typename T>
class Test
{
    public:
    void print()
    {
        cout << "General template object" << endl;
    }
};

template<>   // 对int型特例化
class Test<int>
{
    public:
```

```
    void print()
    {
        cout << "Specialized template object" << endl;
    }
};

```

另外，**与函数模板不同，类模板的特例化不必为所有模板参数提供实参。**我们可以只指定一部分而非所有模板参数，这种叫做类模板的偏特化 或部分特例化（partial specialization）。例如，C++标准库中的类vector的定义：

```
template <typename T, typename Allocator>
class vector
{
    /*.....*/
};

// 部分特例化
template <typename Allocator>
class vector<bool, Allocator>
{
    /*.....*/
};
```

在vector这个例子中，一个参数被绑定到bool类型，而另一个参数仍未绑定需要由用户指定。注意，一个类模板的部分特例化版本仍然是一个模板，因为使用它时用户还必须为那些在特例化版本中未指定的模板参数提供实参。

参考链接：https://blog.csdn.net/lisonglisonglisong/article/details/38057367

其中有一些不对的地方，看的时候需要注意！！！

参考：

一些面试题1-30题： https://blog.csdn.net/weixin_43778179/article/details/90236445

牛客网上的题：https://blog.csdn.net/erwugumo/article/details/89219168

面试常见题：https://blog.csdn.net/qq_22642239/article/details/104964543

面试常见题：https://blog.csdn.net/u012864854/article/details/79777991