1. 谈谈你对C++三个特点的理解（简单描述一下）。

面向对象的三个基本特征是：**封装、继承、多态**。

封装

封装最好理解了。封装是面向对象的特征之一，是对象和类概念的主要特性。

封装，也就是**把客观事物封装成抽象的类**，并且类可以把自己的数据和方法只让可信的类或者对象操作，对不可信的进行信息隐藏。**数据+操作方法**

作用：隐藏细节，使代码模块化

继承

- 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。
- 通过继承创建的新类称为“子类”或“派生类”。
- 被继承的类称为“基类”、“父类”或“超类”。
- 继承的过程，就是从一般到特殊的过程。
- 要实现继承，可以通过“继承”（Inheritance）和“组合”（Composition）来实现。

  作用：扩展已实现的代码模块

多态

多态性（polymorphisn）是允许你将父对象设置成为和一个或更多的他的子对象相等的技术，赋值之后，父对象就可以根据当前赋值给它的子对象的特性以不同的方式运作。简单的说，就是一句话：**允许将子类类型的指针赋值给父类类型的指针**。

多态有两种实现方式：**覆盖和重载**。

但是，重载并不属于面向对象编程。它在编译前就确定了，是静态的。覆盖是晚绑定，动态的。

作用：封装和继承的作用都是代码重用，而多态的作用是接口重用。为了类在继承和派生的时候，保证使用“家谱”中任一类的实例的某一属性时的正确调用。 **一个接口，多种方法**

2. 为什么要封装，如何解释他的用意

权限

3. C++多态的实现机制

C++的多态性用一句话概括就是：**在基类的函数前加上virtual关键字**，在派生类中重写该函数，运行时将会根据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.

- 用***virtual***关键字申明的函数叫做虚函数，虚函数肯定是类的成员函数。
- 存在虚函数的类都有一个一维的虚函数表叫做虚表。类的对象有一个指向虚表开始的虚指针。虚表是和类对应的，虚表指针是和对象对应的。
- 多态性是一个接口多种实现，是面向对象的核心。分为类的多态性和函数的多态性。
- 多态用虚函数来实现，结合动态绑定。
- **纯虚函数**是虚函数再加上= 0。
- **抽象类**是指包括至少一个纯虚函数的类。

4. 纯虚函数的概念，有什么作用

4.1 什么时候需要函数？

- 类会不会作为基类？
- 看子类会不会修改这个方法？

  应考虑对成员函数的调用是通过对象名还是通过基类指针或引用去访问，如果是通过基类指针或引用去访问的，则应当声明为虚函数。有时在定义虚函数时，并不定义其函数体，即**纯虚函数**。它的作用只是定义了一个虚函数名,具体功能留给派生类去添加。

说明：使用虚函数，系统要有一定的空间开销。当一个类带有虚函数时，编译系统会为该类构造一个**虚函数表**（vtbl），它是一个指针数组，存放每个虚函数的入口地址。系统在进行动态关联的时间开销很少，提高了多态性的效率。

4.2 虚析构函数

析构函数的作用就是在对象撤销之前把类对象从内存中撤销。**通常只会执行基类的析构函数，不会执行派生类的析构函数**。虚拟构函数可以解决这个问题。

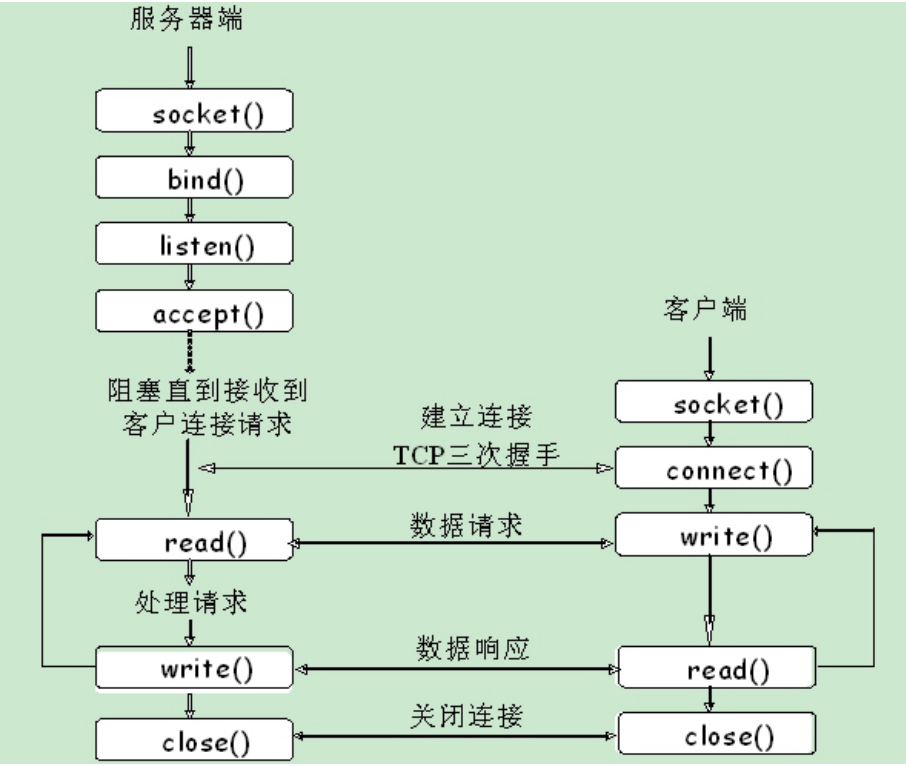
但是构造函数不能是虚函数，因为它不能虚，需要知道构造成什么！

友元函数的优点和缺点

1. 优点：可以使定义为友元函数或者友元类的函数直接访问另一个类中的私有成员和受保护成员，提高效率，方便编程。
2. 缺点：破坏了类的封装性，提供了安全漏洞

5. TCP/IP中服务器端和客户端的实现流程

TCP客户端和服务端程序设计



6. 服务器端是如何处理客户端的连接请求

略

7. 线程与进程的区别

进程

进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动,进程是系统进行资源分配和调度的一个独立单位.

线程

线程是进程的一个实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位.线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.

区别

1) 简而言之,一个程序至少有一个进程,一个进程至少有一个线程.

2) 线程的划分尺度小于进程，使得多线程程序的并发性高。

3) 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

4) 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

5) 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是**进程和线程的重要区别**。

为什么要使用线程？

1．**耗时的操作使用线程，提高应用程序响应**
2．并行操作时使用线程，如C/S架构的服务器端并发线程响应用户的请求。
3．多CPU系统中，使用线程提高CPU利用率
4．改善程序结构。一个既长又复杂的进程可以考虑分为多个线程，成为几个独立或半独

8. 进程间是如何通讯的（在Windows中）

- **管道(pipe)**：管道是一种半双工的通信方式，数据只能单向流动，而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。
- **有名管道 (named pipe)**： 有名管道也是半双工的通信方式，但是它允许无亲缘关系进程间的通信。
- **信号量(semophore)**： 信号量是一个计数器，可以用来控制多个进程对共享资源的访问。它常作为一种锁机制，防止某进程正在访问共享资源时，其他进程也访问该资源。因此，主要作为进程间以及同一进程内不同线程之间的同步手段。
- **消息队列(message queue)**： 消息队列是由消息的链表，存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。
- **信号 (sinal)**： 信号是一种比较复杂的通信方式，用于通知接收进程某个事件已经发生。
- **共享内存(shared memory)**：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。
- **套接字(socket)**： 套解口也是一种进程间通信机制，与其他通信机制不同的是，它可用于不同及其间的进程通信。

9. 线程的同步方式有哪几种？

- **临界区（CCriticalSection）**
  当多个线程访问一个独占性共享资源时，可以使用临界区对象。拥有临界区的线程可以访问被保护起来的资源或代码段，其他线程若想访问，则被挂起，直到拥有临界区的线程放弃临界区为止。具体应用方式是：
  1、定义临界区对象CcriticalSection g_CriticalSection;
  2、在访问共享资源（代码或变量）之前，先获得临界区对象，g_CriticalSection.Lock（）；
  3、访问共享资源后，则放弃临界区对象，g_CriticalSection.Unlock（）；
- **事件(CEvent)**
  事件机制，则允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务。比如在某些网络应用程序中，一个线程如A负责侦听通信端口，另外一个线程B负责更新用户数据，利用事件机制，则线程A可以通知线程B何时更新用户数据。
- **互斥量（CMutex）**
  互斥对象和临界区对象非常相似，只是其允许在进程间使用，而临界区只限制与同一进程的各个线程之间使用，但是更节省资源，更有效率。
- **信号量（CSemphore）**
  当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使用“信号量”对象。CSemaphore类对象保存了对当前访问某一个指定资源的线程的**计数值**，该计数值是当前还可以使用该资源的线程数目。如果这个计数达到了零，则所有对这个CSemaphore类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零为止。

10. 隐藏

相对于重写和重载而言，隐藏发生于基类和派生类之间（区分重载），派生类方法隐藏基类方法：
1. 派生类与基类的方法同名，但是参数不同
2. 派生类与基类的方法同名，参数也相同，而且没有virtual关键字(和重写区分)
3. 如果基类里就有重载多种方法，而派生类只有某一种，那基类其他方法不可用

11. TCP和UDP的区别

很多文章都说TCP协议可靠，UDP协议不可靠！为什么前者可靠，后者不可靠呢？既然UDP协议不可靠，为什么还要使用它呢？所谓的TCP协议是面向连接的协议，面向连接是什么呢？

TCP和UDP都是传输层的协议！从编程的角度看，就是两个模块（模块就是代码的集合，一系列代码的组合提供相应的功能！模块化最终目的就是：分工协作！模块化好处：便于扩展开发以及维护！）。

TCP协议：

这个协议，是**面向的连接**！面向连接这个概念，我们要从物理层看起。大家都知道，因为“信道复用技术”的迅猛发展，才促使了计算机网络的发展！如果没有“信道复用技术”，那么单条线路上（这里的线路指物理传输介质，例如：双绞线、光纤、电话线）单位时间内只能供一台计算机使用！还是举例说明：就拿你自己的计算机来说，你跟同学“小明”聊天的时候，就不能跟另外一位同学“小强”聊天，如果你想同时跟两位同学聊天，那么你就得装两条线路！那么同时与第三位、第四位同学。。。第N位同学聊天的时候，你需要装几根线路？全世界人民聊天的时候，又需要装几根线路？

“信道复用技术”实现了在同一条线路上，单位时间内可供X台计算机同时通信！Toad知道以下几种复用技术：

1、频分复用 2、时分复用 3、波分复用 4、码分复用 5、空分复用 6、统计复用 7、极化波复用

关于“信道复用技术”更深层次的问题，需要你自己去研究！

上面我们提到了“信道复用技术”！知道了这一点，我们就很容易明白“物理信道”上的“虚拟信道”概念了！不同的信道复用技术，使用不同的复用技术，目的就是创建“虚拟信道”。

TCP协议连接其实就是在物理线路上创建的一条“虚拟信道”。这条“虚拟信道”建立后，在TCP协议发出FIN包之前（两个终端都会向对方发送一个FIN包），是不会释放的。正因为这一点，TCP协议被称为面向连接的协议！

UDP协议，一样会在物理线路上创建一条“虚拟信道”，否则UDP协议无法传输数据！但是，当UDP协议传完数据后，这条“虚拟信道”就被立即注销了！因此，称UDP是不面向连接的协议！

个人理解：TCP会告诉对方，“我收到你的消息了”，而UDP直管发，不管是否收到。TCP代价大，要处理拥塞控制、数据校验等问题，不适合实时通信。

12. C++中的类型安全

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

> （1）操作符new返回的指针类型严格与对象匹配，而不是void；
> *（2）C中很多以void为参数的函数可以改写为C++模板函数，而模板是支持类型检查的；*
> （3）引入const关键字代替#define constants，它是有类型、有作用域的，而#define constants只是简单的文本替换；
> （4）一些#define宏可被改写为inline函数，结合函数的重载，可在类型安全的前提下支持多种类型，当然改写为模板也能保证类型安全；
> （5）C++提供了**dynamic*cast****关键字，使得转换过程更加安全，因为*dynamic*cast比static_cast涉及更多具体的类型检查。 即便如此，C++也不是绝对类型安全的编程语言。如果使用不得当，同样无法保证类型安全。比如下面两个例子：

```
int i=5;
void* pInt=&i;
double d=(*(double*)pInt);
cout<
```

输入结果不是5，而意想不到的结果：-9.25596e+061。又比如：

```
+ expand source
```

结果如下：

```
5
1717986918
请按任意键继续. . .
```

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

MFC中CString是类型安全的吗？

不是，其他数据类型转换到CString可以使用CString的成员函数Format方法来实现。

13. C++中为什么使用模板类？

（1）可用来创建动态增长和减小的数据结构
（2）它是类型无关的，因此具有很高的**可复用性**。
（3）它在编译时而不是运行时检查数据类型，保证了类型安全
（4）它是平台无关的，可移植性
（5）可用于基本数据类型

14. 信号与信号量的区别

信号：

信号（signal）是一种处理异步事件的方式。信号时比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程外，还可以发送信号给进程本身。linux除了支持unix早期的信号语义函数，还支持语义符合posix.1标准的信号函数sigaction。

信号量：

信号量（Semaphore）进程间通信处理同步互斥的机制。是在多线程环境下使用的一种设施, 它负责协调各个线程, 以保证它们能够正确、合理的使用公共资源。

---

15. static的作用

C++的static有两种用法：面向过程程序设计中的static和面向对象程序设计中的static。前者应用于普通变量和函数，不涉及类；后者主要说明static在类中的作用。

1. 面向过程设计中的static

静态全局变量

- 该变量在全局数据区分配内存；
- 未经初始化的静态全局变量会被程序自动初始化为0（自动变量的值是随机的，除非它被显式初始化）；
- 静态全局变量在声明它的整个文件都是可见的，而在文件之外是不可见的；

静态局部变量

- 该变量在全局数据区分配内存；
- 静态局部变量在程序执行到该对象的声明处时被首次初始化，即以后的函数调用不再进行初始化；
- 静态局部变量一般在声明处初始化，如果没有显式初始化，会被程序自动初始化为0；
- 它始终驻留在全局数据区，直到程序运行结束。但其作用域为局部作用域，当定义它的函数或语句块结束时，其作用域随之结束；

静态函数

- 在函数的返回类型前加上static关键字,函数即被定义为静态函数。静态函数与普通函数不同，它只能在声明它的文件当中可见，不能被其它文件使用。
- 其它文件中可以定义相同名字的函数，不会发生冲突；

2. 面向对象程序设计中的static

静态数据成员

- 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作是类的成员。无论这个类的对象被定义了多少个，**静态数据成员在程序中也只有一份拷贝**，由该类型的所有对象共享访问。也就是说，静态数据成员是该类的所有对象所共有的。对该类的多个对象来说，静态数据成员只分配一次内存，供所有对象共用。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；

- 静态数据成员**存储在全局数据区**。静态数据成员定义时要分配空间，所以不能在类声明中定义。在Example 5中，语句int Myclass::Sum=0;是定

义静态数据成员；

- 静态数据成员和普通数据成员一样遵从public,protected,private访问规则；
- 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，在没有产生类对象时其作用域就可见，即**在没有产生类的实例时，我们就可以操作它**；
- 静态数据成员初始化与一般数据成员初始化不同。静态数据成员初始化的格式为：
  ＜数据类型＞＜类名＞::＜静态数据成员名＞=＜值＞
- 类的静态数据成员有两种访问形式：
  ＜类对象名＞.＜静态数据成员名＞
  或 ＜类类型名＞::＜静态数据成员名＞ 如果静态数据成员的访问权限允许的话（即public的成员），可在程序中，按上述格式来引用静态数据成员
- 静态数据成员主要用在各个对象都有相同的某项属性的时候。比如对于一个存款类，每个实例的利息都是相同的。所以，应该把利息设为存款类的静态数据成员。这有两个好处，第一，不管定义多少个存款类对象，利息数据成员都共享分配在全局数据区的内存，所以节省存储空间。第二，一旦利息需要改变时，只要改变一次，则所有存款类对象的利息全改变过来了；

同全局变量相比，使用静态数据成员有两个**优势**：
1. 静态数据成员没有进入程序的全局名字空间，因此不存在与程序中其它全局名字冲突的可能性；
2. 可以实现信息隐藏。静态数据成员可以是private成员，而全局变量不能；

静态成员函数

- 出现在类体外的函数定义不能指定关键字static；
- 静态成员之间可以相互访问，包括静态成员函数访问静态数据成员和访问静态成员函数；
- 非静态成员函数可以任意地访问静态成员函数和静态数据成员；
- 静态成员函数不能访问非静态成员函数和非静态数据成员；
- 由于没有this指针的额外开销，因此静态成员函数与类的全局函数相比速度上会有少许的增长；
- 调用静态成员函数，可以用成员访问操作符(.)和(->)为一个类的对象或指向类对象的指针调用静态成员函数，也可以直接使用如下格式：
  ＜类名＞::＜静态成员函数名＞（＜参数表＞） 调用类的静态成员函数。

---

16. 指针和引用的区别？

1. 引用必须被初始化，指针不必；
2. 引用初始化后不能改变，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。

---

17. 虚函数表的存储位置

我们都知道，虚函数是多态机制的基础，就是在程序在运行期根据调用的对象来判断具体调用哪个函数，现在我们来说说它的具体实现原理，主要说一下我自己的理解，如果有什么不对的地方请指正。

在每个包含有虚函数的类的对象的最前面(是指这个对象对象内存布局的最前面，至于为什么是最前面，说来话长，这里就不说了，主要是考虑到效率问题)都有一个称之为虚函数指针(vptr)的东西指向虚函数表(vtbl)，这个虚函数表(这里仅讨论最简单的单一继承的情况，若果是多重继承，可能存在多个虚函数表)里面存放了这个类里面所有虚函数的指针，当我们要调用里面的函数时通过查找这个虚函数表来找到对应的虚函数，这就是虚函数的实现原理。

这里我假设大家都了解了，如果不了解可以去查下资料。好了，既然我们知道了虚函数的实现原理，虚函数指针vptr指向虚函数表vtbl，而且vptr又在对象的最前面，那么我们很容易可以得到虚函数表的地址，下面我写了一段代码测试了一下：

```
```
#include <iostream>
#include <stdio.h>
typedef void (*fun_pointer)(void);

using namespace std;
class Test
{
 public:
```
```

```
   Test()
       {
     cout<<"Test()."<<endl;
    }
   virtual void print()
       {
     cout<<"Test::Virtual void print1()."<<endl;
    }
   virtual void print2()
       {
     cout<<"Test::virtual void print2()."<<endl;
    }
};

class TestDrived:public Test
{
 public:
 static int var;
  TestDrived()
    {
     cout<<"TestDrived()."<<endl;
    }
  virtual void print()
    {
       cout<<"TestDrived::virtual void print1()."<<endl;
    }
  virtual void print2()
    {
    cout<<"TestDrived::virtual void print2()."<<endl;
    }
  void GetVtblAddress()
    {
         cout<<"vtbl address:"<<(int*)this<<endl;
    }
  void GetFirstVtblFunctionAddress()
    {
    cout<<"First vbtl funtion address:"<<(int*)*(int*)this+0 << endl;
    }
  void GetSecondVtblFunctionAddress()
    {
    cout<<"Second vbtl funtion address:"<<(int*)*(int*)this+1 << endl;
    }
  void CallFirstVtblFunction()
    {
    fun = (fun_pointer)* ( (int*) *(int*)this+0 );
     cout<<"CallFirstVbtlFunction:"<<endl;
     fun();
    }
  void CallSecondVtblFunction()
    {
    fun = (fun_pointer)* ( (int*) *(int*)this+1 );
     cout<<"CallSecondVbtlFunction:"<<endl;
     fun();
    }
private:
    fun_pointer fun;
};


int TestDrived::var = 3;

int main()
{
 cout<<"sizeof(int):"<<sizeof(int)<<"sizeof(int*)"<<sizeof(int*)<<endl;
 fun_pointer fun = NULL;
 TestDrived a;
 a.GetVtblAddress();
```

```
 cout<<"The var's address is:"<<&TestDrived::var<<endl;
 a.GetFirstVtblFunctionAddress();
 a.GetSecondVtblFunctionAddress();
 a.CallFirstVtblFunction();
 a.CallSecondVtblFunction();
 return 0;
}
```

18. C++模板库中sort算法的原理

标准模版库中sort函数包含在头文件algorithm中，std::sort()

```
default (1)
template <class RandomAccessIterator>
void sort (RandomAccessIterator first, RandomAccessIterator last);
custom (2)
template <class RandomAccessIterator, class Compare>
  void sort (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

STL的sort()算法，数据量大时采用Quick Sort，分段递归排序，一旦分段后的数据量小于某个门槛，为避免Quick Sort的递归调用带来过大的额外负荷，就改用Insertion Sort。如果递归层次过深，还会改用Heap Sort。下文先分别介绍这个三个Sort，再整合分析STL sort算法(以上三种算法的综合) -- Introspective Sorting(内省式排序)。
STL sort源码剖析

---

19. 动态链接库和静态链接库的区别，及其原理

两者都是代码共享的方式。

静态库：在链接步骤中，连接器将从库文件取得所需的代码，复制到生成的可执行文件中，这种库称为静态库，其特点是可执行文件中包含了库代码的一份完整拷贝；缺点就是被多次使用就会有多份冗余拷贝。即静态库中的指令都全部被直接包含在最终生成的 EXE 文件中了。在vs中新建生成静态库的工程，编译生成成功后，只产生一个.lib文件

动态库：动态链接库是一个包含可由多个程序同时使用的代码和数据的库，DLL不是可执行文件。动态链接提供了一种方法，使进程可以调用不属于其可执行代码的函数。函数的可执行代码位于一个 DLL 中，该 DLL 包含一个或多个已被编译、链接并与使用它们的进程分开存储的函数。在vs中新建生成动态库的工程，编译成功后，产生一个.lib文件和一个.dll文件。

区别：
1. **静态库中的lib**：该LIB包含函数代码本身（即包括函数的索引，也包括实现），在编译时直接将代码加入程序当中
2. **动态库中的lib**：该LIB包含了函数所在的DLL文件和文件中函数位置的信息（索引），函数实现代码由运行时加载在进程空间中的DLL提供。
总之，lib是编译时用到的，dll是运行时用到的。如果要完成源代码的编译，只需要lib；如果要使动态链接的程序运行起来，只需要dll。

---

20. 虚函数和纯虚函数的区别

1. 类里声明虚函数的作用是为了能让这个函数在它的子类里面被覆盖，这样编译器就可以使用后期绑定来达到多态。纯虚函数接口，是个函数的声明而已，它要留到子类里面去实现。
2. **虚函数在子类里面也可以不重载，但是纯虚函数必须在子类里面去实现**。通常，很多函数加上virtual修辞，虽然牺牲掉一些性能，但是增加了面向对象的多态性，可以阻止父类里面的这个函数在子类里被修改实现；
3. 虚函数的类继承了接口，同时也继承了父类的实现。纯虚函数关注的是接口的统一性，实现由子类完成；
4. 带纯虚函数的类叫做虚基类。这种基类不能直接声称对象，只能被继承，并重写其虚函数后才能使用，这样的类也叫抽象类。

---

21. malloc和free,new与delete的使用？

- delete会调用对象的析构函数，new调用构造函数。
- malloc与free是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。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。

- What the huck!