1、类的对象存储空间？

非静态成员的数据类型大小之和。

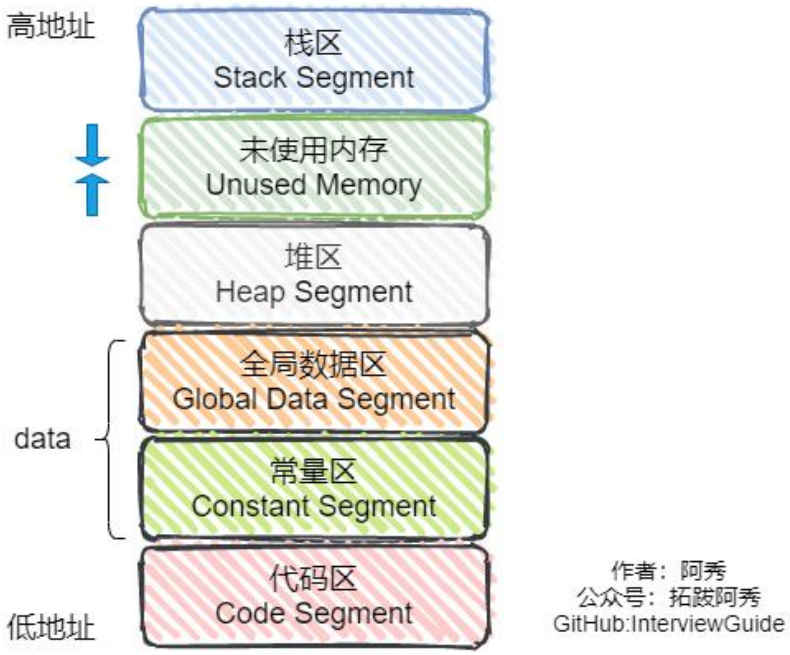
编译器加入的额外成员变量（如指向虚函数表的指针）。

为了边缘对齐优化加入的 padding。

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

2、简要说明 C++的内存分区

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



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

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

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

全局/静态存储区：全局变量和静态变量被分配到同一块内存中，在以前的 C 语言中，全局变量和静态变量又分为初始化的和未初始化的，在 C++里面没有这个区分了，它们共同占用同一块内存区，在该区定义的变量若没有初始化，则会被自动初始化，例如 int 型变量自动初始为 0

常量存储区：这是一块比较特殊的存储区，这里面存放的是常量，不允许修改

代码区：存放函数体的二进制代码

3、什么是内存池，如何实现

内存池（Memory Pool） 是一种**内存分配**方式。通常我们习惯直接使用 new、malloc 等申请内存，这样做的缺点在于：由于所申请内存块的大小不定，当频繁使用时会造成大量的内存碎片并进而降低性能。内存池则是在真正使用内存之前，先申请分配一定数量的、大小相等(一般情况下)的内存块留作备用。当有新的内存需求时，就从内存池中分出一部分内存块，若内存块不够再继续申请新的内存。这样做的一个显著优点是尽量避免了内存碎片，使得内存分配效率得到提升。

这里**简单描述一下《STL 源码剖析》中的内存池实现机制：**

allocate 包装 malloc，deallocate 包装 free

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

1. 首先客户端会调用 malloc()配置一定数量的区块（固定大小的内存块，通常为 8 的倍数），假设 40 个 32bytes 的区块，其中 20 个区块（一半）给程序实际使用，1 个区块交出，另外 19 个处于维护状态。剩余 20 个（一半）留给内存池，此时一共有（20*32byte）
2. 客户端之后有有内存需求，想申请（20*64bytes）的空间，这时内存池只有（20*32bytes），就先将（10*64bytes)个区块返回，1 个区块交出，另外 9 个处于维护状态，此时内存池空空如也.
3. 接下来如果客户端还有内存需求，就必须再调用 malloc()配置空间，此时新申请的区块数量会增加一个随着配置次数越来越大的附加量，同样一半提供程序使用，另一半留给内存池。申请内存的时候用永远是先看内存池有无剩余，有的话就用上，然后挂在 0-15 号某一条链表上，要不然就重新申请。
4. 如果整个堆的空间都不够了，就会在原先已经分配区块中寻找能满足当前需求的区块数量，能满足就返回，不能满足就向客户端报 bad_alloc 异常

allocator 就是用来分配内存的，最重要的两个函数是 allocate 和 deallocate，就是用来申请内存和回收内存的，外部（一般指容器）调用的时候只需要知道这些就够了。

内部实现，目前的所有编译器都是直接调用的::operator new()和::operator delete()，说白了就是和直接使用 new 运算符的效果是一样的，所以老师说它们都没做任何特殊处理。

其实最开始 GC2.9 之前

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

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

GC2.9 下的 alloc 函数的一个比较好的分配器的实现规则如下：

维护一条 0-15 号的一共 16 条链表，其中 0 号表示 8 bytes ，1 号表示 16 bytes，2 号表示 24 bytes。。。。而 15 号表示 16* 8 = 128 bytes。

如果在申请内存时，申请内存的大小并不是 8 的倍数（比如 2、4、7、9、18 这样不是 8 的倍数），那就找刚好能满足内存大小的链表。比如想申请 12 个大小，那就按照 16 来处理，也就是找 1 号链表了；想申请 20 ，距离它最近的就是 24 了，那就找 2 号链表。

只许比所要申请的内容大，不许小！

但是现在 GC4.9 及其之后 也还有 alloc 函数，只不过已经变成_pool_alloc 这个名字了，名字已经改了，也不再是默认的了。

你需要自己手动去指定它可以自己指定，比如

```
vector<string,__gnu_cxx::pool_alloc<string>> vec;
```

这样来使用它，等于兜兜转转又回到以前那种对 malloc 和 free 的包装形式了。

4、可以说一下你了解的 C++得内存管理吗？

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

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

5、C++中类的数据成员和成员函数内存分布情况

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

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

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

class Person
{
public:
    Person()
    {
        this->age = 23;
    }
    void printAge()
    {
        cout << this->age <<endl;
    }
    ~Person(){}
public:
    int age;
};

int main()
{
    Person p;
    cout << "对象地址："<< &p <<endl;
    cout << "age地址："<< &(p.age) <<endl;
    cout << "对象大小："<< sizeof(p) <<endl;
    cout << "age大小："<< sizeof(p.age) <<endl;
    return 0;
}
```

```
//输出结果
//对象地址：0x7fffec0f15a8
//age地址：0x7fffec0f15a8
//对象大小：4
//age大小：4
```

从代码运行结果来看，对象的大小和对象中数据成员的大小是一致的，也就是说，成员函数不占用对象的内存。这是因为所有的函数都是存放在代码区的，不管是全局函数，还是成员函数。

要是成员函数占用类的对象空间，那么将是多么可怕的事情：定义一次类对象就有成员函数占用一段空间。

我们再来补充一下静态成员函数的存放问题：**静态成员函数与一般成员函数的唯一区别就是没有 this 指针**，因此不能访问非静态数据成员。

就像我前面提到的，**所有函数都存放在代码区，静态函数也不例外。所有有人一看到 static 这个单词就主观的认为是存放在全局数据区，那是不对的。**

6、关于 this 指针你知道什么？全说出来

this 指针是类的指针，指向对象的首地址。

this 指针只能在成员函数中使用，在全局函数、静态成员函数中都不能用 this。

this 指针只有在成员函数中才有定义，且存储位置会因编译器不同有不同存储位置。

this 指针的用处

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

this 指针的使用

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

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

类的 this 指针有以下特点

(1）this 只能在成员函数中使用，全局函数、静态函数都不能使用 this。实际上，**传入参数为当前对象地址，成员函数第一个参数为**为 **T * const this**

如：

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

其中，**func** 的原型在编译器看来应该是：

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

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

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

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

7、几个 this 指针的易混问题

A. this 指针是什么时候创建的？

this 在成员函数的开始执行前构造，在成员的执行结束后清除。

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

B. this 指针存放在何处？堆、栈、全局变量，还是其他？

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

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

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

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

D. this 指针是如何访问类中的变量的？

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

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

this 是类的指针，如果换成结构体，那 this 就是结构的指针了。

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

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

F.每个类编译后，是否创建一个类中函数表保存函数指针，以便用来调用函数？

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

8、 内存泄漏的后果？如何监测？解决方法？

1) 内存泄漏

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

2) 后果

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

3) 如何排除

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

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

4) 解决方法

智能指针。

5) 检查、定位内存泄漏

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

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

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

定位代码位置

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

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

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

10、为什么是不可预期的问题？

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

11、 如果在类的析构函数中调用 delete this，会发生什么？

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

12、你知道空类的大小是多少吗？

1. C++空类的大小不为 0，不同编译器设置不一样，vs 设置为 1；
2. C++标准指出，不允许一个对象（当然包括类对象）的大小为 0，不同的对象不能具有相同的地址；
3. 带有虚函数的 C++类大小不为 1，因为每一个对象会有一个 vptr 指向虚函数表，具体大小根据指针大小确定；
4. C++中要求对于类的每个实例都必须有独一无二的地址,那么编译器自动为空类分配一个字节大小，这样便保证了每个实例均有独一无二的内存地址。

13、请说一下以下几种情况下，下面几个类的大小各是多少？

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

空类的大小是 1， 在 C++中空类会占一个字节，这是为了让对象的实例能够相互区别。具体来说，空类同样可以被实例化，并且每个实例在内存中都有独一无二的地址，因此，编译器会给空类隐含加上一个字节，这样空类实例化之后就会拥有独一无二的内存地址。当该空白类作为基类时，该类的大小就优化为 0 了，子类的大小就是子类本身的大小。这就是所谓的空白基类最优化。

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

```
class A { virtual void Fun(){} };int main(){
  cout<<sizeof(A)<<endl;// 输出 4(32 位机器)/8(64 位机器);
  A a;
  cout<<sizeof(a)<<endl;// 输出 4(32 位机器)/8(64 位机器);
  return 0;
}
```

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

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

静态成员存放在静态存储区，不占用类的大小，普通函数也不占用类大小

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

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

14、this 指针调用成员变量时，堆栈会发生什么变化？

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

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

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

15、类对象的大小受哪些因素影响？

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