C++研发高频笔/面试题目  
**笔试：**  
笔试这一块想不到特别好说的，很多时候只是公司筛选人的第一步，分数高低代表意义不大（尤其是大公司，现身说法，我的腾讯笔试真的是呵呵哒），当然笔试是一个门槛，不过后面很可能就不用谈了。我笔试经验也不足，感觉笔试考察最重点的就是算法的熟悉程度，代码能力，我笔试成绩通常不太高，原因是我编程题速度比较慢，往往都做不完。  
以C++开放举例，粗略说说笔试一般会考哪些内容：  
（1） 语言基础，C++中的定义，指针，模板，多态，重载等（尤其是多态，给定一个程序，基类子类有不同实现，考虑输出结果，这一类的题目面试题也极为高频）

**面试高频题：**  
校招过程中参考过牛客诸位大佬的面经，但是具体哪一块是参考谁的我也忘记了，如果您看到有些熟悉，那么可能就是您的，如果您觉得侵权，请告诉我，我撤回。  
1语言基础（C++）  
（1） 指针和引用的区别

★ 相同点：

    1. 都是地址的概念；

    指针指向一块内存，它的内容是所指内存的地址；引用是某块内存的别名。

    ★ 区别：

    1. 指针是一个实体，而引用仅是个别名；

    2. 引用使用时无需解引用（\*），指针需要解引用；

    3. 引用只能在定义时被初始化一次，之后不可变；指针可变；

    引用“从一而终” ^\_^

    4. 引用没有 const，指针有 const，const 的指针不可变；

    5. 引用不能为空，指针可以为空；

    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（所指向的变量或对象的地址）的大小；

    typeid（T） == typeid（T&） 恒为真，sizeof（T） == sizeof（T&） 恒为真，但是当引用作为成员时，其占用空间与指针相同（没找到标准的规定）。

    7. 指针和引用的自增（++）运算意义不一样；

    ★ 联系

    1. 引用在语言内部用指针实现（如何实现？）。

    2. 对一般应用而言，把引用理解为指针，不会犯严重语义错误。引用是操作受限了的指针（仅容许取内容操作）。

    引用是[C++](http://c.chinaitlab.com/)中的概念，初学者容易把引用和指针混淆一起。一下程序中，n 是m 的一个引用（reference），m是被引用物（referent）。

    int m；

    int &n = m；

    n 相当于m 的别名（绰号），对n 的任何操作就是对m 的操作。例如有人名叫王小毛，他的绰号是“三毛”。说“三毛”怎么怎么的，其实就是对王小毛说三道四。所以n 既不是m 的拷贝，也不是指向m 的指针，其实n就是m 它自己。

    引用的一些规则如下：

    （1）引用被创建的同时必须被初始化（指针则可以在任何时候被初始化）。

    （2）不能有NULL 引用，引用必须与合法的[存储](http://www.storworld.com/)单元关联（指针则可以是NULL）。

    （3）一旦引用被初始化，就不能改变引用的关系（指针则可以随时改变所指的对象）。

    以下示例程序中，k 被初始化为i 的引用。语句k = j 并不能将k 修改成为j 的引用，只是把k 的值改变成为6.由于k 是i 的引用，所以i 的值也变成了6.

    int i = 5；

    int j = 6；

    int &k = i；

    k = j； // k 和i 的值都变成了6；

    上面的程序看起来象在玩文字游戏，没有体现出引用的价值。引用的主要功能是传递函数的参数和返回值。[C++](http://c.chinaitlab.com/)语言中，函数的参数和返回值的传递方式有三种：值传递、指针传递和引用传递。

    以下是“值传递”的示例程序。由于Func1 函数体内的x是外部变量n 的一份拷贝，改变x 的值不会影响n， 所以n 的值仍然是0.

|  |
| --- |
| void Func1(int x) { x = x + 10; } int n = 0; Func1(n); cout << “n = ” << n << endl;// n = 0 |

    以下是“指针传递”的示例程序。由于Func2 函数体内的x 是指向外部变量n 的指针，改变该指针的内容将导致n 的值改变，所以n 的值成为10.

|  |
| --- |
| void Func2(int \*x) { (\* x) = (\* x) + 10; } &#8943; int n = 0; Func2(&n); cout << “n = ” << n << endl; // n = 10 |

    以下是“引用传递”的示例程序。由于Func3 函数体内的x 是外部变量n 的引用，x和n 是同一个东西，改变x 等于改变n，所以n 的值成为10.

|  |
| --- |
| void Func3(int &x) { x = x + 10; } &#8943; int n = 0; Func3(n); cout << “n = ” << n << endl; // n = 10 |

对比上述三个示例程序，会发现“引用传递”的性质象“指针传递”，而书写方式象“值传递”。实际上“引用”可以做的任何事情“指针”也都能够做，为什么还要“引用”

    这东西？

    答案是“用适当的工具做恰如其分的工作”。

    指针能够毫无约束地操作内存中的如何东西，尽管指针功能强大，但是非常危险。

    就象一把刀，它可以用来砍树、裁纸、修指甲、理发等等，谁敢这样用？

    如果的确只需要借用一下某个对象的“别名”，那么就用“引用”，而不要用“指针”，以免发生意外。比如说，某人需要一份证明，本来在文件上盖上公章的印子就行了，如果把取公章的钥匙交给他，那么他就获得了不该有的权利。

    ——————————

    摘自「高质量c＋＋编程」

    指针与引用，在MoreEffective C++ 的条款一有详细讲述，我给你转过来

    条款一：指针与引用的区别

    指针与引用看上去完全不同（指针用操作符‘\*’和‘->’，引用使用操作符‘。’），但是它们似乎有相同的功能。指针与引用都是让你间接引用其他对象。你如何决定在什么时候使用指针，在什么时候使用引用呢？

    首先，要认识到在任何情况下都不能用指向空值的引用。一个引用必须总是指向某些对象。因此如果你使用一个变量并让它指向一个对象，但是该变量在某些时候也可能不指向任何对象，这时你应该把变量声明为指针，因为这样你可以赋空值给该变量。相反，如果变量肯定指向一个对象，例如你的设计不允许变量为空，这时你就可以把变量声明为引用。

    “但是，请等一下”，你怀疑地问，“这样的代码会产生什么样的后果？”

    char \*pc = 0；// 设置指针为空值

    char& rc = \*pc；// 让引用指向空值

    这是非常有害的，毫无疑问。结果将是不确定的（编译器能产生一些输出，导致任何事情都有可能发生），应该躲开写出这样代码的人除非他们同意改正错误。如果你担心这样的代码会出现在你的软件里，那么你最好完全避免使用引用，要不然就去让更优秀的程序员去做。我们以后将忽略一个引用指向空值的可能性。

    因为引用肯定会指向一个对象，在C里，引用应被初始化。

    string& rs；// 错误，引用必须被初始化

    string s（"xyzzy"）；

    string& rs = s；// 正确，rs指向s

    指针没有这样的限制。

    string \*ps；// 未初始化的指针

    // 合法但危险

    不存在指向空值的引用这个事实意味着使用引用的代码效率比使用指针的要高。因为在使用引用之前不需要[测试](http://softtest.chinaitlab.com/)它的合法性。

|  |
| --- |
| void printDouble(const double& rd) {      cout << rd; // 不需要[测试](http://softtest.chinaitlab.com/)rd,它 } // 肯定指向一个double值 相反，指针则应该总是被测试，防止其为空： void printDouble(const double \*pd) {      if (pd)       { // 检查是否为NULL            cout << \*pd;      } } |

    指针与引用的另一个重要的不同是指针可以被重新赋值以指向另一个不同的对象。但是引用则总是指向在初始化时被指定的对象，以后不能改变。

|  |
| --- |
| string s1("Nancy"); string s2("Clancy"); string& rs = s1; // rs 引用 s1 string \*ps = &s1; // ps 指向 s1 rs = s2; // rs 仍旧引用s1, // 但是 s1的值现在是 // "Clancy" ps = &s2; // ps 现在指向 s2; // s1 没有改变 |

    总的来说，在以下情况下你应该使用指针，一是你考虑到存在不指向任何对象的可能（在这种情况下，你能够设置指针为空），二是你需要能够在不同的时刻指向不同的对象（在这种情况下，你能改变指针的指向）。如果总是指向一个对象并且一旦指向一个对象后就不会改变指向，那么你应该使用引用。

    还有一种情况，就是当你重载某个操作符时，你应该使用引用。最普通的例子是操作符[].这个操作符典型的用法是返回一个目标对象，其能被赋值。

|  |
| --- |
| vector<int> v(10); // 建立整形向量（vector），大小为10; // 向量是一个在标准C库中的一个模板(见条款35) v[5] = 10; // 这个被赋值的目标对象就是操作符[]返回的值 如果操作符[]返回一个指针，那么后一个语句就得这样写： \*v[5] = 10; |

    但是这样会使得v看上去象是一个向量指针。因此你会选择让操作符返回一个引用。（这有一个有趣的例外，参见条款30）

    当你知道你必须指向一个对象并且不想改变其指向时，或者在重载操作符并为防止不必要的语义误解时，你不应该使用指针。而在除此之外的其他情况下，则应使用指针假设你有

|  |
| --- |
| void func(int\* p, int&r); int a = 1; int b = 1; func(&a,b); |

    指针本身的值（地址值）是以passby value进行的，你能改变地址值，但这并不会改变指针所指向的变量的值，

    p = some other pointer；//a is still 1

    但能用指针来改变指针所指向的变量的值，

    \*p = 123131； // a now is 123131

    但引用本身是以pass by reference进行的，改变其值即改变引用所对应的变量的值

    r = 1231；// b now is 1231

    尽可能使用引用，不得已时使用指针。

    当你不需要“重新指向”时，引用一般优先于指针被选用。这通常意味着引用用于类的公有接口时更有用。引用出现的典型场合是对象的表面，而指针用于对象内部。

    上述的例外情况是函数的参数或返回值需要一个“临界”的引用时。这时通常最好返回/获取一个指针，并使用 NULL 指针来完成这个特殊的使命。（引用应该总是对象的别名，而不是被解除引用的NULL 指针）。

    注意：由于在调用者的代码处，无法提供清晰的的引用语义，所以传统的 C 程序员有时并不喜欢引用。然而，当有了一些 C++ 经验后，你会很快认识到这是信息隐藏的一种形式，它是有益的而不是有害的。就如同，程序员应该针对要解决的问题写代码，而不是机器本身。

以下两点以为不妥：

★ 区别：

    3. 引用只能在定义时被初始化一次，之后不可变；指针可变；

    引用“从一而终” ^\_^

    4. 引用没有 const，指针有 const，const 的指针不可变；

证明：

\*/

#include <stdio.h>

#include <iostream>

using namespace std;

/\*

    测试平台：win7 32bit G++

\*/

int main(){

    int i = 2;

    int &a = i; //定义引用必须初始化

    cout << a << endl;

    int &b = i; //引用指向一个变量

    cout << b << endl;

    int j = 12;

    b = j; //引用指向另一个变量，success，引用不是从一而终

    cout << b << endl; //12

    cout << i << endl; //i的值被改为12

    i = 10;

    cout << b << endl; //10,b始终指向i

    cout << i << endl; //i的值被改为10

    cout << j << endl; //j的值为12

    int const &c = i+1; //定义常量引用

    cout << c << endl;

    //c = j; //error,常量引用不能更改

    const int &d = i+2;

    cout << d << endl;

    //d = j; //error,常量引用不能更改

    return 0;

}

/\*

    总结：

    3. 引用只能在定义时被初始化一次，之后可以改变指向的对象，但是仍然指向初始化时的对象；指针可变；

    4. 引用有 const，且从一而终，不可改变指向的对象，指针也有 const，指针常量不可改指向，常量指针可改变指向；

\*/

指针和数组的区别

数组是指针？

最近在做数据结构课设，其中一个函数发生了令人费解的错误，简化后的代码如下：

#include <stdio.h>

int main()

{

char foo[] = "abcde";

char \*\*bar = &foo;

printf("%c\n", \*(\*bar));

return 0;

}

程序执行到 printf 语句后便会挂掉，调试时会提示一个SIGSEGV信号，根据原来的经验，这时程序试图访问本不应该访问的内存。

原来在 C 语言课堂上老师经常提到数组就是一个指针，指针也可以像数组那样用使用中括号的方式来进行内存访问。以这样的想法来分析前面的程序：foo 是一个字符指针，即 foo 的值即为“abcde”的首字符“a”的地址，\*foo 即为 ‘a’；那么 foo 这个指针一定存在某个内存单元，&foo获得这个内存单元的地址，即 pfoo 是指向 foo 的指针，那么\*pfoo 得到 foo，\*(\*pfoo)应该得到‘a’了；这样理解的话，程序是不应该有问题的。

下面我们使用指针代替数组来实现上面的程序：

#include <stdio.h>

#include <stdlib.h>

int main()

{

char \*foo = (char \*)malloc(sizeof (char) \* 2);

\*foo = 'a';

\*(foo + 1) = 0;

char \*\*pfoo = &foo;

printf("%c\n", \*(\*pfoo));

return 0;

}

程序这次运行结果和预料的相同，输出一个字母a。由此可见，数组就是指针，这种说法是错误的。

数组是静态常量指针（static/Compile-time constant）？

有人认为数组是一个静态常量，即数组名代表一个静态的地址值，在编译时确定，下面代码可以证伪这种说法

int main()

{

char foo[] = {'a'};

static char \*p = foo;

return 0;

}

使用 gcc 编译时会有以下错误：

error: initializer element is not constant

可见数组名并不是代表一个静态量，并非地址常量。如果定义 foo 时加上 static 限定符，编译就会通过，此时数组名才代表了一个静态量。

数组是动态常量指针（const/Runtime constant）？

请看以下代码：

int main()

{

char foo[] = {'a'};

char \* const bar; //为什么是这种写法，请自行查阅相关资料

char \*baz;

foo = baz; /\* 1 \*/

bar = baz; /\* 2 \*/

return 0;

}

gcc 编译时错误信息为：

/\* 1 \*/ error: incompatible types when assigning to type 'char[1]' from type 'char \*'

/\* 2 \*/ error: assignment of read-only variable 'bar'

两处出错信息并不相同，若数组为动态常量指针，出错信息应像 2 那样。

数组是什么？

数组既不是静态常量，也不是指针，那么数组是什么？

左值和右值

首先补充一些左值和右值的知识，引用《C专家编程》中的一段话：

出现在赋值符左边的符号有时被称为左值，出现在赋值符右边的符号有时被称为右值。编译器为每个变量分配一个地址（左值）。这个地址在编译时可知，而且该变量在运行时一直保存于这个地址。相反，存储于变量中的值（它的右值）只有在运行时才可知。如果需要用到变量中存储的值，编译器就发出指令从指定地址读入变量值并将它存于寄存器中。

我对左值的理解和书上有些区别，我把这里的“符号”称为“对象”，每一个符号都代表一个对象，对象与地址是一一对应的。即如果声明了 int a，那么 a 作为一个左值时，a 即代表这个保存在某个特定的地址的对象，对这个对象赋值即为把值放在这个特定的地址；a 作为右值时即代表 a 的内容，就是一个单纯的值，而不是对象。一个值是不能作为左值的，比如一个常数 1, 1 = a 这样的赋值语句是无法编译通过的。在我看来，“左值”义同“对象”，“右值”义同“值”，所以下面“左值”和“对象”指的是相同的东西。但是“左值”又有一个子集：“可修改的左值”，只有这个子集中的东西才能放在赋值号左边，因此我认为将引用中的第一句话修改为“出现在赋值符左边的符号有时被称为可修改的左值”更能表达其实际的意思。为什么要引出这个子集，为的就是要把数组分出来，数组是左值，但并不是可修改的左值，因此你也不能直接把数组名放在等号左边进行赋值。

数组就是数组！

我先把结论放在这里，然后在进行分析：数组就是数组，一个数组名就代表一个数组对象，这个对象内可以有一个或多个元素，每个元素类型都相同；正如 int 就是 int，一个 int 变量名就代表一个 int 类型对象。看到这里，你可能要笑了，这不是什么都没说吗，谁不知道数组是这个意思啊，我想知道数组和指针什么关系。其实对数组的认识就是这样一个返璞归真过程，看我来慢慢解释。

以下代码：

/\* 1.c \*/

int main()

{

int foo[] = {1};

int bar = 1;

return 0;

}

使用 gcc 将其汇编并以 intel 格式输出汇编语言文件：

gcc -S -masm=intel 1.c

关键部分为：

mov DWORD PTR [esp+8], 1

mov DWORD PTR [esp+12], 1

esp+8 位置就是那个 int foo[]，esp+12 位置就是那个 int bar。可见，给 int 数组的赋值时就像给一个 int 变量赋值一样，并没用指针来进行间接访问，这个 int 数组对象 foo 的内存地址在编译时就确定了，是 esp+8；正如那个 int 对象 bar 一样，它的内存地址在编译时也确定了，是esp+12。

以示区别，我将下面代码同样以汇编语言输出：

/\* 2.c \*/

#include <stdlib.h>

int main()

{

int \*foo = (int \*)malloc(sizeof (int));

\*foo = 1;

return 0;

}

汇编的关键部分为：

mov DWORD PTR [esp], 4

call \_malloc

mov DWORD PTR [esp+28], eax

mov eax, DWORD PTR [esp+28]

mov DWORD PTR [eax], 1

前两句为 foo 分配内存空间，第三句将分配的内存空间地址值赋给 foo，foo 的地址为 esp+28，编译时已知。下面是赋值部分，首先从 foo 那里得到地址值，然后向这个地址赋值，这里可以看出和给数组赋值的差别，给数组赋值时是将值直接赋到了数组中，而不用从哪里得到数组的地址。

由上面可以看出，数组更像一个普通的变量，编译时就知道了其地址，可以直接赋值。

数组作为左值

数组不能放在赋值号左边，但数组仍可以作为一个左值或者说对象出现在语句中，一个重要的例子就是取地址操作：&。取地址操作 &的操作数必须是一个左值，而不能是一个右值。比如一个变量int a = 1，&a 就可以得到 a 的地址，但 &1 是非法的，一个单纯的数值是没有地址的。那么对于一个int foo[]，&foo 会返回一个什么样的值呢？自然是一个指向数组的指针咯，下面的程序可以看出来：

int main()

{

int foo[1];

int bar[1];

bar = &foo; //故意触发一个 error

return 0;

}

那个赋值语句一定会触发一个的错误，我们可以根据编译输出来确定它们的类型，错误为：

error: incompatible types when assigning to type 'int[1]' from type 'int (\*)[1]'

没错，&foo 返回数据类型为 int (\*)[1]，就是一个指向数组的指针。指向数组？指向数组的哪里呢？指向数组对象首地址，正如一个指向 int 对象的指针指向那个 int 对象占有的两个或四个内存单元的首地址一样。

把 &foo 赋给一个普通的指针是可以的，不过会触发一个 warning，因为int \* 与 int (\*)[1] 并不相容。赋值后普通指针的值与 &foo 的值是相同的，都是数组对象的首地址，只是普通指针把这块内存当做 int 对象处理而已。

由于 C 语言是弱类型语言，你把 &foo 赋给int \*\*\*\*\*\*\*\*\*\*bar 或者 int \*baz都是可以的，都不会导致 error，只会导致 warning，此时你打印出 \*bar 或者 \*baz 的值都是 foo 中第一个整数的值（前提是指针和数组占用空间大小相等）。正如文章开头的代码那样，以这个整数的值作为一个地址值进行间接访问（\*(\*bar)）就会导致非法访问的错误。

数组作为右值

数组作为右值时会发生什么？返回数组对象内的所有值自然不可能，因此 C 语言中采取的方法是数组作为右值时返回对象中元素类型的指针，指针指向第一个元素，类似上一个例子：

int main()

{

int foo[1];

int bar[1];

bar = foo; //故意触发一个 error

return 0;

}

出错信息为：

error: incompatible types when assigning to type 'int[1]' from type 'int \*'

foo 作为右值时返回了一个 int \*，就是这个特性给人造成了数组就是指针的假象。

总结

数组作为左值和数组作为右值时的区别造成了无数人的困惑与误解：foo 作为右值时确实等价于一个指针，因为数组无法像普通对象那样返回它的值，它的元素可能有成百上千个，但作为一个左值时——比如作为取地址操作符的操作数时，数组就是作为一个数组对象而出现的，而不是指针，取地址返回一个指向数组的指针，而不是指向指针的指针。

一句话总结就是：数组就是数组，有着自己的特性。

（题外话：从生成的汇编语言看，用指针来访问内存实际上并不比使用数组来访问内存快，反而是慢了）

转载请注明来源 <http://blog.csdn.net/imred/article/details/45441457>

（2） 堆和栈的区别

一、预备知识—程序的内存分配    
  一个由C/C++编译的程序占用的内存分为以下几个部分    
  1、栈区（stack）—   由编译器自动分配释放  ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈。    
  2、堆区（heap） 一般由程序员分配释放，  若程序员不释放，程序结束时可能由OS回  收   。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，呵呵。    
  3、全局区（静态区）（static）—，全局变量和静态变量的存储是放在一块的，初始化的    
  全局变量和静态变量在一块区域，未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。程序结束后由系统释放。    
  4、文字常量区   —常量字符串就是放在这里的。   程序结束后由系统释放    
  5、程序代码区—存放函数体的二进制代码。    
   
  二、例子程序      
  这是一个前辈写的，非常详细      
  //main.cpp      
  int   a   =   0;   全局初始化区      
  char   \*p1;   全局未初始化区      
  main()      
  {      
  int   b;   栈      
  char   s[]   =   "abc";   栈      
  char   \*p2;   栈      
  char   \*p3   =   "123456";   123456/0在常量区，p3在栈上。      
  static   int   c   =0；   全局（静态）初始化区      
  p1   =   (char   \*)malloc(10);      
  p2   =   (char   \*)malloc(20);      
  分配得来得10和20字节的区域就在堆区。      
  strcpy(p1,   "123456");   123456/0放在常量区，编译器可能会将它与p3所指向的"123456"    
  优化成一个地方。      
  }      
  二、堆和栈的理论知识      
  2.1申请方式      
  stack:   由系统自动分配。  例如，声明在函数中一个局部变量   int   b;   系统自动在栈中为b开辟空间      
  heap:   需要程序员自己申请，并指明大小，在c中malloc函数      
  如p1   =   (char   \*)malloc(10);      
  在C++中用new运算符      
  如p2   =   new   char[10];      
  但是注意p1、p2本身是在栈中的。      
  2.2 申请后系统的响应      
  栈：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。      
  堆：首先应该知道操作系统有一个记录空闲内存地址的链表，当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表  中删除，并将该结点的空间分配给程序，另外，对于大多数系统，会在这块内存空间中的 首地址处记录本次分配的大小，这样，代码中的delete语句才能正确的释放本内存空间。  另外，由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中。      
2.3申请大小的限制      
 栈：在Windows下,栈是向低地址扩展的数据结构，是一块连续的内存的区域。这句话的意思是栈顶的地址和栈的最大容量是系统预先规定好的，在WINDOWS下，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。      
 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。      
2.4申请效率的比较：      
栈由系统自动分配，速度较快。但程序员是无法控制的。      
堆是由new分配的内存，一般速度比较慢，而且容易产生内存碎片,不过用起来最方便. 另外，在WINDOWS下，最好的方式是用VirtualAlloc分配内存，他不是在堆，也不是在栈是直接在进程的地址空间中保留一块内存，虽然用起来最不方便。但是速度快，也最灵活。    
    
  2.5堆和栈中的存储内容      
  栈：在函数调用时，第一个进栈的是主函数中后的下一条指令（函数调用语句的下一条可执行语句）的地址，然后是函数的各个参数，在大多数的C编译器中，参数是由右往左入栈 的，然后是函数中的局部变量。注意静态变量是不入栈的。当本次函数调用结束后，局部变量先出栈，然后是参数，最后栈顶指针指向最开始存的地址，也就是主函数中的下一条指令，程序由该点继续运行。      
堆：一般是在堆的头部用一个字节存放堆的大小。堆中的具体内容由程序员排。      
  2.6存取效率的比较      
  char   s1[]   =   "aaaaaaaaaaaaaaa";      
  char   \*s2   =   "bbbbbbbbbbbbbbbbb";      
  aaaaaaaaaaa是在运行时刻赋值的；      
  而bbbbbbbbbbb是在编译时就确定的；      
  但是，在以后的存取中，在栈上的数组比指针所指向的字符串(例如堆)快。      
  比如：      
  #include      
  void   main()      
  {      
  char   a   =   1;      
  char   c[]   =   "1234567890";      
  char   \*p   ="1234567890";      
  a   =   c[1];      
  a   =   p[1];      
  return;      
  }      
  对应的汇编代码      
  10:   a   =   c[1];      
  00401067   8A   4D   F1   mov   cl,byte   ptr   [ebp-0Fh]      
  0040106A   88   4D   FC   mov   byte   ptr   [ebp-4],cl      
  11:   a   =   p[1];      
  0040106D   8B   55   EC   mov   edx,dword   ptr   [ebp-14h]      
  00401070   8A   42   01   mov   al,byte   ptr   [edx+1]      
  00401073   88   45   FC   mov   byte   ptr   [ebp-4],al      
  第一种在读取时直接就把字符串中的元素读到寄存器cl中，而第二种则要先把指针值读到    
  edx中，再根据edx读取字符，显然慢了。

  2.7小结：      
  堆和栈的区别可以用如下的比喻来看出：      
  使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就 走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。      
使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味而且自由  度大(经典！)

（3） new和delete是如何实现的，new 与 malloc的异同处

**malloc/free与new/delete异同点**

* **相同点**

malloc/free与new/delete都可以用于申请动态内存和释放内存，他们申请的空间都在堆上分配。

* **不同点**

1）操作对象不同

malloc/free是C++/C语言的标准库文件，new/delete是C++的运算符；

对非内部数据对象，malloc/free无法满足动态对象要求。对象在创建时要自动执行构造函数，对象消亡之前要自动执行析构函数，而malloc/free是库函数，不是运算符，故不在编译器控制权限之内，不能够将执行构造函数和析构函数强加于malloc/free身上。而由于new/delete是C++语言，能够完成动态内存分配和初始化工作，并能够完成清理与释放内存工作，即能够自动执行构造函数和析构函数；

2）用法不同

malloc分配内存空间前需要计算分配内存大小；而new能够自动分配内存空间；

malloc是底层函数，其函数返回值类型为void \*；而new运算符调用无参构造函数，故返回值为对应对象的指针；

malloc函数类型不是安全的，编译器不对其进行类型转换、类型安全的相关检查。malloc申请空间后，不会对其初始化，要单独初始化；而new类型是安全的，因为它内置了sizeof、类型转换和类型安全检查功能，且在创建对象时，就完成了初始化工作，一般初始化调用无参构造函数；

operator new对应于malloc，且operator new可以重载，可以自定义内存分配策略，甚至不做内存分配，甚至分配到非内存设备上；但malloc不能。

free只进行释放空间；而delete则释放空间的同时调用析构函数。此外delete使用是注意释放数组的方法为delete []数组名。

* **联系**

new和delete功能覆盖了malloc/free，但因C++程序常会用到C函数，而C函数只能使用malloc/free管理动态内存。此外，使用是malloc和free搭配使用，new和delete搭配使用，不能混乱使用。

（4） C和C++的区别  
（5） C++、Java的联系与区别，包括语言特性、垃圾回收、应用场景等（java的垃圾回收机制）

通常，我们聊到Java，第一印象“面向对象”，“没有指针，编写效率高，执行效率较低”。更深入、专业一点就谈论 “**java内存自动回收(GC垃圾回收机制)**，**多线程编程**”。\*\*   
java的三大特性是封装、继承和多态。\*\*

总结如下：

1、 **JAVA的应用在高层，C++在中间件和底层**

2、 JAVA离不开业务逻辑，而C++可以离开业务为JAVA们服务

3、 java语言给开发人员提供了**更为简洁的语法**；取消了指针带来更高的代码质量；完全面向对象，独特的运行机制是其具有天然的可移植性。

4、 java 是运行在JVM上的，之所以说它的可移植性强，是因为jvm可以安装到任何的系统

5、 c++不是不能在其他系统运行，而是c++在不同的系统上运行，需要不同的编码（**这一点不如java，只编写一次代码，到处运行**）。java程序一般都是生成字节码，在JVM里面运行得到结果。

6、 java 在**web 应用**上具有c++ 无可比拟的优势

7、 java在桌面程序上不如c++实用，**C++可以直接编译成exe文件，指针是c++的优势，可以直接对内存的操作，但同时具有危险性** 。（操作内存的确是一项非常危险的事情，一旦指针指向的位置发生错误，或者误删除了内存中某个地址单元存放的重要数据，后果是可想而知的）。

8、 垃圾回收机制的区别。**c++用析构函数回收垃圾，java自动回收（GC算法）**,写C和C++程序时一定要注意内存的申请和释放。

9、 java 丰富的插件是java 发展如此迅速的原因   
10、 java 很大的沿袭了c++的一些实用结构   
11、 对于**底层程序的编程以及控制方面的编程**，c++很灵活，因为有**句柄**的存在。

Java并不仅仅是C++语言的一个变种，它们在某些本质问题上有根本的不同：

**(1)Java比C++程序可靠性更高**。有人曾估计每50行C++程序中至少有一个BUG。姑且不去讨论这个数字是否夸张，但是任何一个C++程序员都不得不承认C++语言在提供强大的功能的同时也提高了程序含BUG的可能性。Java语言通过改变语言的特性大大提高了程序的可靠性。

**(2)Java语言不需要程序对内存进行分配和回收**。Java丢弃了C++ 中很少使用的、很难理解的、令人迷惑的那些特性，如操作符重载、多继承、自动的强制类型转换。特别地，**Java语言不使用指针，并提供了自动的废料收集**，在Java语言中，内存的分配和回收都是自动进行的，程序员无须考虑内存碎片的问题。

**(3)Java语言中没有指针的概念，引入了真正的数组**。不同于C++中利用指针实现的“伪数组”，Java引入了真正的数组，同时将容易造成麻烦的指针从语言中去掉，这将有利于防止在c++程序中常见的因为数组操作越界等指针操作而对系统数据进行非法读写带来的不安全问题。

**(4)Java用接口(Interface)技术取代C++程序中的多继承性。**接口与多继承有同样的功能，但是省却了多继承在实现和维护上的复杂性。

（6） Struct和class的区别

C++中的struct对C中的struct进行了扩充，它已经不再只是一个包含不同数据类型的数据结构了，它已经获取了太多的功能。  
**struct能包含成员函数吗？ 能！  
struct能继承吗？ 能！！  
struct能实现多态吗？ 能！！！**

既然这些它都能实现，那它和class还能有什么区别？

最本质的一个区别就是默认的访问控制：

**默认的继承访问权限**

struct是public的，class是private的。  
你可以写如下的代码：  
struct A  
{  
  char a;  
}；  
struct B : A  
{  
  char b;  
}；

这个时候B是public继承A的。

如果都将上面的struct改成class，那么B是private继承A的。这就是默认的继承访问权限。

所以我们在平时写类继承的时候，通常会这样写：  
class B : public A

就是为了指明是public继承，而不是用默认的private继承。

当然，到底默认是**public继承还是private继承**，**取决于子类而不是基类**。

我的意思是，**struct可以继承class**，同样**class也可以继承struct**，那么默认的继承访问权限是看子类到底是用的struct还是class。如下：

struct A{}；class B : A{}; //private继承  
struct C : B{}； //public继承

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

我依旧强调struct是一种数据结构的实现体，虽然它是可以像class一样的用。我依旧将struct里的变量叫数据，class内的变量叫成员，虽然它们并无区别。

到底是用struct还是class，完全看个人的喜好，你可以将程序里所有的class全部替换成struct，它依旧可以很正常的运行。但我给出的最好建议，还是：当你觉得你要做的更像是一种数据结构的话，那么用struct，如果你要做的更像是一种对象的话，那么用class。

当然，我在这里还要强调一点的就是，对于访问控制，应该在程序里明确的指出，而不是依靠默认，这是一个良好的习惯，也让你的代码更具可读性。

说到这里，很多了解的人或许都认为这个话题可以结束了，因为他们知道struct和class的“唯一”区别就是访问控制。很多文献上也确实只提到这一个区别。

但我上面却没有用“唯一”，而是说的“最本质”，那是因为，它们确实还有另一个区别，虽然那个区别我们平时可能很少涉及。

那就是：“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。这一点在Stanley B.Lippman写的Inside the C++ Object Model有过说明。

问题讨论到这里，基本上应该可以结束了。但有人曾说过，他还发现过其他的“区别”，那么，让我们来看看，这到底是不是又一个区别。还是上面所说的，C++中的struct是对C中的struct的扩充，既然是扩充，那么它就要兼容过去C中struct应有的所有特性。例如你可以这样写：

struct A //定义一个struct  
{  
   char c1;  
   int n2;  
   double db3;  
};  
A a={'p', 7, 3.1415926}; //定义时直接赋值

也就是说struct可以在定义的时候用{}赋初值。那么问题来了，class行不行呢？将上面的struct改成class，试试看。报错！噢~于是那人跳出来说，他又找到了一个区别。我们仔细看看，这真的又是一个区别吗？

你试着向上面的struct中加入一个构造函数（或虚函数），你会发现什么？  
对，struct也不能用{}赋初值了  
的确，以{}的方式来赋初值，只是用一个初始化列表来对数据进行按顺序的初始化，如上面如果写成A a={'p',7};则c1,n2被初始化，而db3没有。这样简单的copy操作，只能发生在简单的数据结构上，而不应该放在对象上。加入一个构造函数或是一个虚函数会使struct更体现出一种对象的特性，而使此{}操作不再有效。

事实上，是因为加入这样的函数，使得类的内部结构发生了变化。而加入一个普通的成员函数呢？你会发现{}依旧可用。其实你可以将普通的函数理解成对数据结构的一种算法，这并不打破它数据结构的特性。

那么，看到这里，我们发现即使是struct想用{}来赋初值，它也必须满足很多的约束条件，这些条件实际上就是让struct更体现出一种数据机构而不是类的特性。

那为什么我们在上面仅仅将struct改成class，{}就不能用了呢？

其实问题恰巧是我们之前所讲的——访问控制！你看看，我们忘记了什么？对，将struct改成class的时候，访问控制由public变为private了，那当然就不能用{}来赋初值了。加上一个public，你会发现，class也是能用{}的，和struct毫无区别！！！

做个总结，从上面的区别，我们可以看出，struct更适合看成是一个数据结构的实现体，class更适合看成是一个对象的实现体。

（7） define 和const的区别（编译阶段、安全性、内存占用等）

**#define RADIUS 100;**

**const**float   **RADIUS = 100;**

(1) 编译器处理方式不同

　　define宏是在预处理阶段展开。

　　const常量是编译运行阶段使用。

(2) 类型和安全检查不同

　　define宏没有类型，不做任何类型检查，仅仅是展开。

　　const常量有具体的类型，在编译阶段会执行类型检查。

(3) 存储方式不同

　　define宏仅仅是展开，有多少地方使用，就展开多少次，不会分配内存。（宏定义不分配内存，**变量定义**分配内存。）

　　const常量会在内存中分配(可以是堆中也可以是栈中)。

(4)const  可以节省空间，避免不必要的内存分配。 例如：    
        #define PI 3.14159 //常量宏    
        const doulbe Pi=3.14159; //此时并未将Pi放入ROM中 ......    
        double i=Pi; //此时为Pi分配内存，以后不再分配！    
        double I=PI; //编译期间进行宏替换，分配内存    
        double j=Pi; //没有内存分配    
        double J=PI; //再进行宏替换，又一次分配内存！    
        const定义常量从汇编的角度来看，只是给出了对应的内存地址，而不是象#define一样给出的是立即数，所以，const定义的常量在程序**运行过程**中只有一份拷贝（因为是全局的只读变量，存在静态区），而 #define定义的常量在**内存**中有若干个拷贝。   
(5) 提高了效率。 编译器通常不为普通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。

(6) 宏替换只作替换，不做计算，不做表达式求解;

       宏预编译时就替换了，**程序运行时，并不分配内存。**

const 与 #define的比较

    C++ 语言可以用const来定义常量，也可以用 #define来定义常量。但是前者比后者有更多的优点：

（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。

l  **【规则**5-2-1**】**在C++ 程序中只使用const常量而不使用宏常量，即const常量完全取代宏常量。

5.3 常量定义规则

l  **【规则**5-3-1**】**需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

l  **【规则**5-3-2**】**如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

例如：

const  float   RADIUS = 100;

const  float   DIAMETER = RADIUS \* 2;

5.4 类中的常量

有时我们希望某些常量只在类中有效。由于#define定义的宏常量是全局的，不能达到目的，于是想当然地觉得应该用const修饰数据成员来实现。const数据成员的确是存在的，但其含义却不是我们所期望的。const数据成员只在某个对象生存期内是常量，而对于整个类而言却是可变的，因为类可以创建多个对象，不同的对象其const数据成员的值可以不同。

    不能在类声明中初始化const**数据成员**。以下用法是错误的，因为类的对象未被创建时，编译器不知道SIZE的值是什么。

    class A

    {…

        const int SIZE = 100; // 错误，企图在类声明中初始化const数据成员

        int array[SIZE];       // 错误，未知的SIZE

    };

const数据成员的初始化只能在类构造函数的初始化表中进行，例如

    class A

    {…

        A(int size);       // 构造函数

        const int SIZE ;

    };

    A::A(int size) : SIZE(size) // 构造函数的初始化表

    {

      …

    }

    A  a(100);  // 对象 a 的SIZE值为100

    A  b(200);  // 对象 b 的SIZE值为200

    怎样才能建立在整个类中都恒定的常量呢？别指望const数据成员了，应该用类中的枚举常量来实现。例如

    class A

    {…

        enum { SIZE1 = 100, SIZE2 = 200}; //枚举常量

        int array1[SIZE1];

        int array2[SIZE2];

    };

    枚举常量不会占用对象的存储空间，它们在编译时被全部求值。枚举常量的缺点是：它的隐含数据类型是整数，其最大值有限，且不能表示浮点数（如PI=3.14159）。sizeof(A) = 1200;其中枚举部长空间。

           enum   EM { SIZE1 = 100, SIZE2 = 200}; //枚举常量   sizeof（EM） = 4；

（8） 在C++中const和static的用法（定义，用途）  
（9） const和static在类中使用的注意事项（定义、初始化和使用）

**static和const关键字的作用可以从两个方面回答：一是和类的成员函数或者成员变量相关，二是不属于类的函数或者变量。**

**static关键字的作用：**

1、函数体内static变量的作用范围为该函数体，不同于auto变量，该变量的内存只被分配一次，因此，其值在下次调用的时候仍然维持原始值。

2、在模块内的static全局变量可以被模块内的所有函数访问，但是不能被模块外的其他函数访问。

3、在模块内的static函数只可以被这一模块内的其他函数调用，这个函数的使用范围被限制在声明它的模块内。

4、在类中的static成员变量属于整个类所有，对类的所有对象只有一份拷贝。

5、在类中的static成员函数属于整个类所有，这个函数不接受this指针，因而只能访问类的static成员变量。

**const关键字的作用：**

1、想要阻止一个变量被改变，可以使用const关键字。在定义该const关键字是，通常要对它进行初始化，因为以后再也没有机会去改变它。

2、对于指针来说，可以指定指针本省为const，也可以指定指针所指向的数据为const，或者二者同时指定为const。

3、在一个函数声明中，const可以修饰形参，表明它是一个输入参数，在函数内部不能改变其值。

4、对于类的成员函数，若指定为const，则表明其实一个常函数，不能修改类的成员变量。

5、对于类的成员函数，有时候必须制定其返回值为const，以使得其返回值不能为左值。

（10） C++中的const类成员函数（用法和意义）  
（11） 计算下面几个类的大小：  
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;  
（12） 给一个代码，求输出结果  
class A  
{  
public:  
A(int x){}  
}  
问：A a = 1;是否正确, 如果正确, 那么它调用了哪些函数？  
这类题目更常见的是在基类和子类有不同实现方法。（虚函数相关，栗子很多，不多说了）

（13） C++的STL介绍（这个系列也很重要，建议侯捷老师的这方面的书籍与视频），其中包括内存管理allocator，函数，实现机理，多线程实现等

（14） STL源码中的hash表的实现

（15） STL中unordered\_map和map的区别

（16） STL中vector的实现

（17） vector使用的注意点及其原因，频繁对vector调用push\_back()对性能的影响和原因。

（18） C++中的重载和重写的区别：

（19） C ++内存管理（热门问题）

（20） 介绍面向对象的三大特性，并且举例说明每一个。

（21） 多态的实现（和下个问题一起回答）

（22） C++虚函数相关（虚函数表，虚函数指针），虚函数的实现原理（热门，重要）

<http://blog.csdn.net/haoel/article/details/1948051/>

（23） 实现编译器处理虚函数表应该如何处理

（24） 析构函数一般写成虚函数的原因

（25） 构造函数为什么一般不定义为虚函数

（26） 构造函数或者析构函数中调用虚函数会怎样

（27） 纯虚函数  
（28） 静态绑定和动态绑定的介绍

（29） 引用是否能实现动态绑定，为什么引用可以实现

（30） 深拷贝和浅拷贝的区别（举例说明深拷贝的安全性）

（31） 对象复用的了解，零拷贝的了解

（32） 介绍C++所有的构造函数

（33） 什么情况下会调用拷贝构造函数（三种情况）

（34） 结构体内存对齐方式和为什么要进行内存对齐？

（35） 内存泄露的定义，如何检测与避免？

（36） 手写实现智能指针类（34-37我没遇见过）

（37） 调试程序的方法

（38） 遇到coredump要怎么调试

（39） 内存检查工具的了解

（40） 模板的用法与适用场景

（41） 成员初始化列表的概念，为什么用成员初始化列表会快一些（性能优势）？

（42） 用过C11吗，知道C11新特性吗？（有面试官建议熟悉C11）

（43） C++的调用惯例（简单一点C++函数调用的压栈过程）

（44） C++的四种强制转换  
 类型转换有c风格的，当然还有c++风格的。c风格的转换的格式很简单（TYPE）EXPRESSION，但是c风格的类型转换有不少的缺点，有的时候用c风格的转换是不合适的，因为它可以在任意类型之间转换，比如你可以把一个指向const对象的指针转换成指向非const对象的指针，把一个指向基类对象的指针转换成指向一个派生类对象的指针，这两种转换之间的差别是巨大的，但是传统的c语言风格的类型转换没有区分这些。还有一个缺点就是，c风格的转换不容易查找，他由一个括号加上一个标识符组成，而这样的东西在c++程序里一大堆。所以c++为了克服这些缺点，引进了4新的类型转换操作符，他们是1.static\_cast  2.const\_cast  3.dynamic\_cast  4.reinterpret\_cast.

1.static\_cast

最常用的类型转换符，在正常状况下的类型转换，如把int转换为float，如：int i；float f； f=（float）i；或者f=static\_cast<float>(i);

2.const\_cast

用于取出const属性，把const类型的指针变为非const类型的指针，如：const int \*fun(int x,int y){}　　int \*ptr=const\_cast<int \*>(fun(2.3))

3.dynamic\_cast

该操作符用于运行时检查该转换是否类型安全，但只在多态类型时合法，即该类至少具有一个虚拟方法。dynamic\_cast与static\_cast具有相同的基本语法，dynamic\_cast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。如：

class C

{  
　　//…C没有虚拟函数  
}；  
class T{  
　　//…  
}  
int main()  
{  
　　dynamic\_cast<T\*> (new C);//错误  
}  
此时如改为以下则是合法的：  
class C

{  
public:  
　　virtual void m() {};// C现在是 多态  
}

4.reinterpret\_cast

interpret是解释的意思，reinterpret即为重新解释，此标识符的意思即为数据的二进制形式重新解释，但是不改变其值。如：int i; char \*ptr="hello freind!"; i=reinterpret\_cast<int>(ptr);这个转换方式很少使用。

6设计模式和算法  
设计模式一般都不会考太多，除非你明确说自己懂。我基本上就不涉及到设计模式的东西，所以只是简要说说。当然，单例模式和简单工厂模式的概念和使用场景还是要知道的。  
（1） 介绍熟悉的设计模式（单例，简单工厂模式）  
（2） 写单例模式（饿汉模式和懒汉模式），线程安全版本  
（3） MVC设计模式  
算法这一块太过庞大，几乎都有可能，牛油们最好还是去刷剑指offer（level 1），leetcode（如果能够刷到最高难度，算法对你来说已经不是什么了，或者说面试对你来说简直就是吃饭喝水的难度），左神的书《程序源代码面试指南》（字符串，数组，dp，海量数据问题，搞定它们也就搞定面试的一半）。  
下面还是简单的列举一些吧（包括一些数据结构题目，只列举简单的，面试的算法一半不会太难，但是现在一般都是需要比较好的思维，或者曾经接触过这方面的题，建议就是多刷题，做题感觉是刷出来的）  
（1） 红黑树的了解（平衡树，二叉搜索树），使用场景  
（2） 红黑树在STL上的应用  
（3） 了解并查集吗？（低频）  
（4） 贪心算法和动态规划的区别  
（5） 判断一个链表是否有环，如何找到这个环的起点  
（6） 实现一个strcpy函数（或者memcpy），如果内存可能重叠呢  
（7） 实现一个循环队列  
（8） 排序算法（写快排，归并排序，堆排序），算法的时间复杂度，空间复杂度，是否稳定等  
（9） 快排存在的问题，如何优化  
（10） 反转一个链表  
（11） Top K问题（可以采取的方法有哪些，各自优点？）  
（12） Bitmap的使用，存储和插入方法  
（13） 二叉树的先序、中序、后序遍历（非递归实现）  
（14） 二叉树的公共祖先（简单地说，剑指offer上的题大都是高频题）  
（15） 1-n中有多少个1  
（16） 字典树的理解以及在统计上的应用  
（17） 数组的全排列  
（18） N个骰子出现和为m的概率  
（19） 海量数据问题（可参考左神的书）  
（20） 一致性哈希  
7智力题  
我特别把智力题单独拿出来讲，是因为这一块其实有些公司很看重（例如腾讯），但是我基本上也没刷过这类题目，接触不多，牛油们自行补充吧。  
（1） 100层楼，只有2个鸡蛋，想要判断出那一层刚好让鸡蛋碎掉，给出策略（滴滴笔试中两个铁球跟这个是一类题）  
（2） 毒药问题，n拼毒药，要快速找出哪一瓶有毒，需要几只小白鼠  
（3） 博弈论问题  
（4） 先手必胜策略问题：n本书，每次能够拿X-X本，怎么拿必胜  
（5） 放n只蚂蚁在一条树枝上，蚂蚁与蚂蚁之间碰到就各自往反方向走，问总距离或者时间。  
（6） 瓶子换饮料问题：多少个瓶子能够换1瓶饮料，问最多能喝几瓶